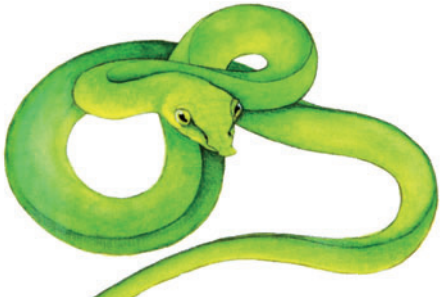
CAY HORSTMANN

CONCEITOS DE COMPUTAÇÃO COM

Compatível com
Java 5 & 6

Java

5ª Edição



bookman®

Sobre o autor

CAY S. HORSTMANN é professor de ciência da computação no departamento de ciência da computação da Universidade Estadual de San Jose. É experiente programador, foi vice-presidente e gerente de tecnologia da Preview Systems, Inc. Presta consultoria em C++, Java, Windows e programação Internet para importantes corporações, universidades e organizações. Horstmann é autor de muitos livros profissionais e acadêmicos bem-sucedidos, incluindo *Padrões e Projetos Orientados a Objetos*, *Big Java*, *Conceitos de Computação com o Essencial de C++* (publicados pela Bookman Editora sob esses títulos), *Big C++* e *Core Java*, com Gary Cornell.

CÓPIA NÃO AUTORIZADA É CRIME
ABDR
ASSOCIAÇÃO BRASILEIRA DE DIREITOS REPROGRÁFICOS
RESPEITE O DIREITO AUTORAL

H819c Horstmann, Cay.
Conceitos de computação em Java [recurso eletrônico] / Cay Horstmann ; tradução Edson Furmankiewicz. – 5. ed. – Dados eletrônicos – Porto Alegre : Bookman, 2009.

Editado também como livro eletrônico em 2009.
Conteúdo: Capítulos 16, 17, 18 e apêndices de D a M disponíveis em: www.bookman.com.br.
ISBN 978-85-7780-407-8

1. Computação – Linguagem de programação. I. Título.

CDU 004.438JAVA

Catalogação na publicação: Renata de Souza Borges CRB-10/Prov-021/08

CAY HORSTMANN
SAN JOSE STATE UNIVERSITY

CONCEITOS DE COMPUTAÇÃO COM Java

Compatível com
Java 5 & 6

5ª Edição

Tradução:
Edson Furmankiewicz

Consultoria, supervisão e revisão técnica desta edição:
Luciana Porcher Nedel
Doutora em Ciências pela École Polytechnique Fédérale de Lausanne, Suíça
Professora do Instituto de Informática da UFRGS

Versão impressa
desta obra: 2009

bookman®

2009

Obra originalmente publicada sob o título
Java Concepts for Java 5 and 6, 5th Edition
ISBN 978-0-470-10555-9

Copyright © 2008 John Wiley & Sons, Inc. All rights reserved.
This translation published under license.

Capa: *Rogério Grilho, arte sobre capa original*

Preparação de original: *Verônica de Abreu Amaral*

Supervisão editorial: *Denise Weber Nowaczyk*

Editoração eletrônica: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
ARTMED® EDITORA S.A.
(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S. A.)
Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

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

SÃO PAULO
Av. Angélica, 1.091 – Higienópolis
01227-100 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Prefácio

Este livro é um texto introdutório à ciência da computação, com foco nos princípios e nas práticas da programação. Por que você deve escolher este livro para o seu primeiro curso em ciência da computação? Eis as principais razões:

- O ponto de vista do livro vai além da sintaxe da linguagem, focando nos conceitos da ciência da computação.
- O paradigma orientado a objetos é enfatizado desde o início, expondo-o a objetos e classes nos primeiros capítulos para que você não precise desaprender hábitos procedurais.
- Em vez de seguir uma abordagem rígida do tipo "objetos desde o início" ou "objetos só mais tarde", os conceitos de orientação a objetos são introduzidos gradualmente.
- Um foco no desenvolvimento baseado em testes encoraja o leitor a testar seus programas à medida que os desenvolve, proporcionando maior confiança em relação a suas soluções, se estão corretas ou não.
- O livro motivará o leitor a dominar os aspectos práticos da programação, a partir de uma grande quantidade de dicas úteis sobre as boas práticas da engenharia de software.
- O livro ensina a linguagem Java padrão – não um ambiente especializado como uma "bicicleta com rodinhas de segurança para o iniciante não cair". A linguagem, a biblioteca e as ferramentas Java são apresentadas em profundidade suficiente para resolver problemas de programação do mundo real.

O uso de Java

Este livro se baseia na linguagem de programação Java. Java foi escolhido por quatro razões:

- Orientação a objeto
- Segurança
- Simplicidade
- Amplitude da biblioteca padrão

A orientação a objetos é o paradigma predominante no projeto de software. A orientação a objetos permite aos programadores investir mais tempo projetando seus programas e menos tempo codificando e depurando. Este livro começa com objetos e classes desde o início para que os leitores não precisem desaprender hábitos de programação procedural mais tarde.

Segurança é uma importante característica da linguagem Java e altamente benéfica para programadores iniciantes. Erros de programação comuns são confiavelmente diagnosticados quando um programa Java é compilado ou executado. Utilizando Java, você terá mais tempo para investir no aprimoramento de projetos de softwares substanciais em vez de gastá-lo depurando programas com comportamentos misteriosos e irreproduzíveis.

Uma outra vantagem importante de Java é sua simplicidade. É possível dominar as construções essenciais da linguagem em um curso de um semestre.

Por fim, a biblioteca Java padrão tem uma amplitude suficiente para ser utilizada em várias disciplinas de um curso de ciência da computação. Elementos gráficos, construção de interfaces com o usuário, acesso a bancos de dados, multithreading e programação de redes são parte da biblioteca padrão. Portanto, as habilidades aprendidas em uma disciplina introdutória baseada em Java servirão muito bem para disciplinas subseqüentes.

Novo nesta edição

Desenvolvimento baseado em testes

A metodologia de desenvolvimento baseado em testes promete o desenvolvimento rápido de softwares livres de defeitos evitando as entediantes sessões de depuração. O princípio central é desenvolver casos de teste juntamente com a, ou mesmo antes da, implementação do comportamento desejado.

Em um ambiente de produção, o desenvolvimento baseado em testes é auxiliado por um conjunto de bibliotecas e ferramentas que podem ser difíceis para o iniciante. Neste livro, focamos no aspecto essencial do desenvolvimento baseado em testes – ou seja, pensar nos resultados esperados ao desenvolver um programa. Exigimos dos programas de teste nada mais do que imprimir o resultado esperado junto com qualquer saída do programa. Uma exigência assim tão simples encoraja o leitor a pensar mais profundamente sobre seus programas à medida que os desenvolve.

Há testes de verificação de aprendizagem distribuídos por vários capítulos, iniciando no Capítulo 2. Os tópicos vão desde o desenvolvimento de programas testadores simples até a aplicação de testes de regressão e o uso de um depurador.

Objetos introduzidos gradualmente

Há muita discussão na comunidade da ciência da computação se a programação orientada a objetos deveria ser introduzida "desde o início" ou "só mais tarde". Nenhuma abordagem é a ideal. A abordagem "objetos introduzidos gradualmente" utilizada neste livro foi ainda mais refinada nesta edição. No Capítulo 2, você aprenderá a utilizar objetos e classes com biblioteca padrão. No Capítulo 3, você aprenderá a implementar classes a partir de uma especificação dada. Esses dois capítulos permitem que você utilize objetos convenientemente ao estudar os principais tópicos algorítmicos nos Capítulos 4 a 7 – sem aprender maus hábitos que você precisará desaprender mais tarde. No Capítulo 8, você aprenderá como projetar a especificação de uma classe. Por fim, no Capítulo 12, você aprenderá como descobrir classes desenvolvidas por colaboradores para desenvolver soluções orientadas a objetos para problemas complexos de programação.

Atualizado para Java 6

Java 6, distribuído em dezembro de 2006, contém algumas melhorias úteis na biblioteca, principalmente para interfaces com o usuário, bancos de dados e programação XML. O livro foi atualizado a fim de abranger essas melhorias. O material básico não é dependente de Java 6, e o livro pode ser utilizado igualmente com Java 5 ou Java 6.

Seção opcional com exercícios gráficos

Fornecemos seções opcionais com exercícios gráficos porque muitos estudantes gostam de escrever programas que criem desenhos e porque formas gráficas são ótimos exemplos de objetos. Todas as seções opcionais de exercícios gráficos podem ser desconsideradas se desejado.

Nesta edição, a cobertura dos elementos gráficos foi dividida em partes menores. O Capítulo 2 contém material sobre desenhos simples e o Capítulo 3 descreve como implementar objetos que se autodesenham. O Capítulo 9, que introduz a noção de interface, usa programação baseada em eventos para reforçar o conceito de interface. O Capítulo 10 cobre herança e a seção sobre elementos gráficos mostra como utilizar herança para personalizar áreas de desenho. Por fim, o Capítulo 18 cobre componentes de interfaces gráficas com o usuário e seus leiaute.

Integração com a Web

Para aumentar o foco nos conceitos e reduzir o tamanho do livro impresso, parte do material opcional e avançado foi disponibilizado na Web. O material presente apenas na Web é identificado pelo ícone ⊕ e pode ser acessado pelo site www.bookman.com.br.

WileyPLUS

O pacote WileyPLUS contém material suplementar, está em inglês e é de inteira responsabilidade da editora original. O WileyPLUS pode ser adotado por instrutores que desejam que seus alunos tirem o máximo proveito de um amplo espectro de ferramentas on-line para prática e aprofundamento no assunto, lições de casa e questio-

nários, animações e projetos de laboratório. O WileyPLUS integra todos os recursos Web para instrutores e alunos em uma versão on-line deste texto. Para informações adicionais e uma demonstração, visite o site www.wiley.com/college/wileyplus. Todo conteúdo on-line que é parte integrante do livro está disponível, em português, no site da Bookman Editora, www.bookman.com.br.

LabRat: uma ferramenta para programação de exercícios práticos

Todos os exercícios de programação nesta edição foram revisados para permitir que a avaliação possa ser auxiliada por computador. Especificações mais detalhadas e código para que iniciantes aprendam a atribuir aos exercícios estão disponíveis no WileyPLUS. Uma ferramenta inovadora dentro do WileyPLUS, o LabRat, permite que instrutores programem exercícios do tipo "complete o código" e programação e os disponibilizem on-line. Os alunos podem utilizar o LabRat para verificar se suas soluções estão de acordo com as especificações do programa antes de as submeterem para avaliação on-line. O LabRat fornece aos instrutores um relatório com os resultados do teste, o código dos alunos e as notas recomendadas que o instrutor pode atribuir ou substituir no livro de notas do WileyPLUS.

Recursos para o aluno e para o professor no WileyPLUS

Os seguintes recursos para alunos e professores podem ser encontrados no curso WileyPLUS para este texto:

- Soluções para todos os exercícios (apenas para professores)
- Um banco de teste (apenas para professores)
- Um manual de laboratório
- Exercícios adicionais de revisão e programação
- Uma lista de perguntas feitas com freqüência
- Ajuda com compiladores mais comuns
- Slides de apresentação em aula que resumem cada capítulo e incluem listagens de código e figuras do livro
- O código-fonte de todos os exemplos do livro
- O guia de estilo de programação em formato eletrônico, para que você possa modificá-lo e adequá-lo a preferências locais
- Um guia de conversão para migrar da edição anterior (apenas para professores)

BlueJ junto com o livro

Há também um manual do BlueJ (disponível nos Estados Unidos) e que foi personalizado especificamente para ser utilizado com este livro. O manual vem com um CD-ROM que contém o software BlueJ e todos os exemplos de código deste livro, formatados como projetos BlueJ.

Um tour pelo livro

O livro pode ser naturalmente agrupado em quatro partes, como ilustrado pela Figura 1. A organização dos capítulos oferece a mesma flexibilidade da edição anterior; dependências entre os capítulos também são mostradas na figura.

Parte A: Fundamentos (Capítulos 1 a 7)

O Capítulo 1 contém uma breve introdução à ciência da computação e à programação Java. O Capítulo 2 mostra como manipular objetos de classes predefinidas. No Capítulo 3, você construirá suas próprias classes simples a partir das especificações dadas.

Tipos de dados fundamentais, desvios, laços e arranjos são cobertos nos Capítulos 4 a 7.

Parte B: Projeto orientado a objetos (Capítulos 8 a 12)

O Capítulo 8 discute o projeto de classes de uma maneira sistemática e introduz um subconjunto muito simples da notação UML.

A discussão sobre polimorfismo e herança foi dividida em dois capítulos. O Capítulo 9 cobre interfaces e polimorfismo, enquanto o Capítulo 10 cobre herança. Introduzir interfaces antes da herança tem uma vantagem importante: os alunos vêem imediatamente o polimorfismo antes de se verem em dificuldades com detalhes técnicos como a construção de superclasses.

O tratamento de exceções e entrada/saída básica a partir de arquivo são abordados no Capítulo 11. A hierarquia de exceções fornece um exemplo útil de herança.

O Capítulo 12 contém uma introdução a um projeto orientado a objetos, incluindo dois estudos de caso significativos.

Parte C: Uma introdução a estruturas de dados e a algoritmos (Capítulos 13 a 15)

Os Capítulos 13 a 15 contêm uma introdução a algoritmos e estruturas de dados básicas, cobrindo recursividade, classificação e pesquisa, listas encadeadas, pilhas e filas. Esses tópicos podem estar além do escopo de uma disciplina de um semestre, mas podem ser abordados conforme desejado após o Capítulo 7 (veja Figura 1).

A recursividade é introduzida a partir de um ponto de vista orientado a objetos: Um objeto que resolve um problema recursivamente constrói outro objeto da mesma classe que resolve um problema mais simples. A idéia de fazer outro objeto resolver o trabalho mais simples é mais intuitiva do que implementar uma função que chama a si mesma.

O Capítulo 14 cobre os algoritmos de classificação fundamentais e fornece uma introdução sutil à análise big-Oh. O Capítulo 15 introduz listas encadeadas, pilhas e filas como tipos de dados abstratos, como elas aparecem na biblioteca Java padrão.

Parte D: Tópicos Avançados (Capítulos 16 a 18)

O Capítulo 16 cobre estruturas de dados avançadas: tabelas hash, árvores binárias de pesquisa e heaps. O Capítulo 17 introduz generalidades de Java. Esse capítulo é adequado

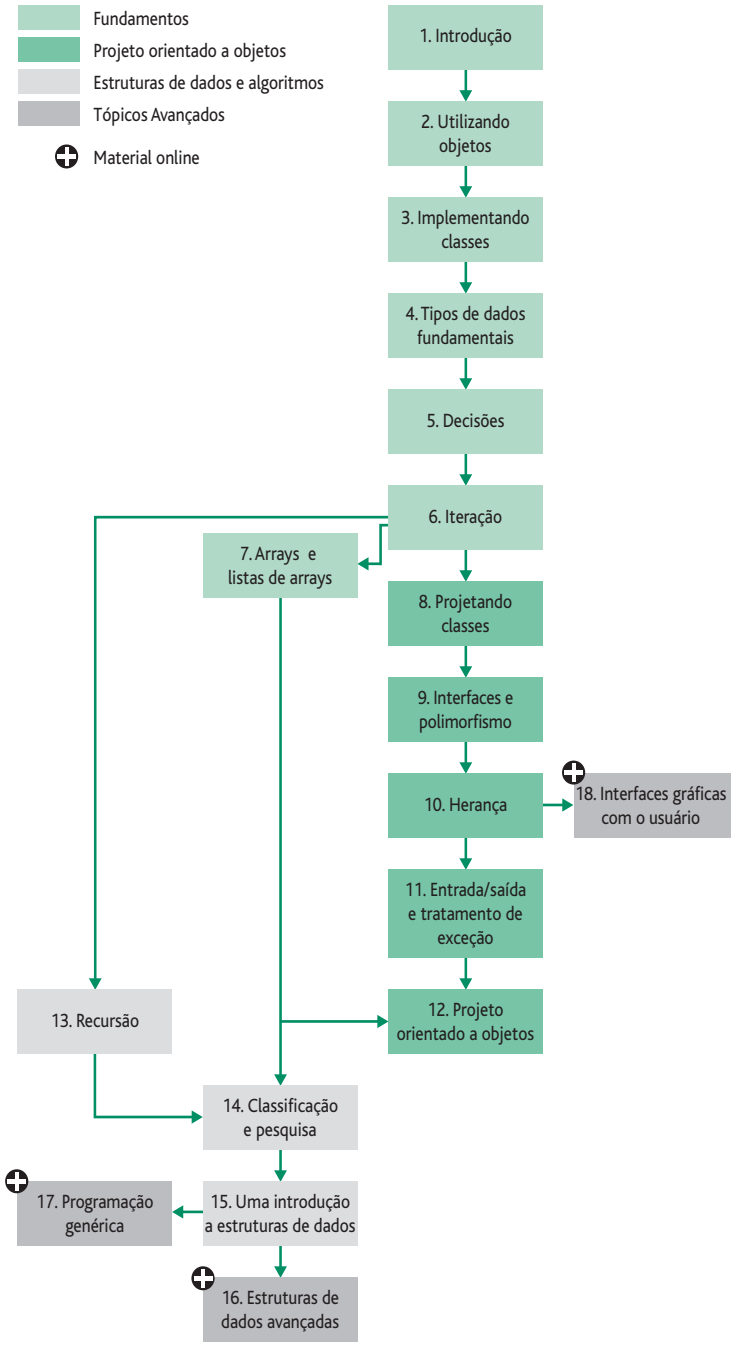


Figura 1 Dependências entre os capítulos.

para alunos avançados que querem implementar suas próprias classes e métodos genéricos. O Capítulo 18 conclui a série de exercícios gráficos sobre interfaces com o usuário discutindo o gerenciamento de leiautes e os componentes Swing. Esses capítulos estão disponíveis na Web, no material online do livro, no site www.bookman.com.br.

Apêndices

O Apêndice A contém um guia de estilo para ser utilizado com este livro. Muitos professores acreditam que é altamente benéfico a exigência de um estilo consistente para todos os exercícios. Mas se esse guia de estilo entrar em conflito com as opiniões do professor ou com os costumes locais, ele também está disponível em formato eletrônico para que possa ser modificado. O Apêndice B lista os subconjuntos Latino Básico e Latino-1 do Unicode. O Apêndice C contém uma visão geral das classes e interfaces na biblioteca padrão abordada neste livro. Apêndices adicionais que contêm referências rápidas sobre a sintaxe Java, HTML, ferramentas Java, números binários, UML e outros materiais adicionais estão disponíveis no site www.bookman.com.br.

A estrutura pedagógica

Esta edição aprimora os elementos pedagógicos usados na última edição e oferece auxílio adicional ao leitor. Cada capítulo começa com a visão geral habitual dos objetivos do capítulo e com uma introdução motivacional. Em seguida, uma lista do conteúdo de cada capítulo fornece uma referência rápida aos recursos especiais no capítulo.

Notas destacadas marcam e reforçam novos conceitos e são resumidas no final do capítulo.

Em todos os capítulos, notas destacadas indicam onde novos conceitos são introduzidos e fornecem uma descrição das idéias-chave. Essas notas são resumidas como uma revisão no final de cada capítulo.

Cada seção termina com uma série de exercícios de autoverificação da aprendizagem. Utilize esses exercícios para verificar se você entendeu os tópicos introduzidos. Os exercícios de autoverificação não são triviais – eles foram propositadamente projetados para fazer você pensar por meio do novo material. As respostas aos exercícios de autoverificação estão no final de cada capítulo.

As listagens dos programas foram cuidadosamente projetadas para uma leitura fácil, indo muito além da simples codificação em cores. Os comentários são compostos em uma fonte diferente, mais fácil de ler do que a fonte monoespaçada de “computador”. Os métodos são identificados por um fundo branco. Palavras-chave, strings e números são codificados em cores. [O código para todas as listagens de programa do livro (além de todos os arquivos adicionais necessários para cada exemplo) está disponível na Web em www.bookman.com.br.]

Ao longo de todos os capítulos, elementos gráficos especiais realçam tópicos a fim de permitir maior flexibilidade na leitura e facilitar referências. As caixas de sintaxe destacam novas construções sintáticas e seus propósitos. A lista dessas construções em ordem alfabética pode ser encontrada na página 21.

Seis elementos gráficos, intitulados “Erros Comuns”, “Como Fazer”, “Dica de Produtividade”, “Dica de Qualidade”, “Tópico Avançado” e “Fato Aleatório”, identificados pelos ícones a seguir, aparecem separados do corpo do texto para não interromper o fluxo de leitura do assunto principal. Alguns desses elementos são bem curtos; outros se estendem por uma página inteira. Cada tópico recebeu o espaço necessário para uma explicação completa e convincente – em vez de ser espremido em uma “dica” de um parágrafo. Você pode utilizar as tabelas nas próximas páginas para consultar a lista desses elementos em cada capítulo e os números de página onde eles se encontram.

- **Erros Comuns** descrevem os tipos de erro que estudantes cometem com freqüência, com uma explicação do motivo desses erros e o que fazer em relação a eles. A maioria dos estudantes descobre rapidamente as seções *Erros comuns* e as lê por conta própria.

- **Como Fazer** são seções inspiradas nos guias HOWTO do Linux. Essas seções destinam-se a responder a perguntas de iniciantes do tipo “O que eu faço agora?”, fornecendo instruções passo a passo para tarefas comuns.

- **Dicas de Qualidade** explicam as boas práticas de programação. Como a maioria dessas práticas exige maior investimento de trabalho inicial, essas notas explicam o motivo da sugestão e mostram como o esforço será recompensado mais tarde.

- **Dicas de Produtividade** ensinam os estudantes a utilizar ferramentas de modo mais eficiente. Muitos iniciantes não prestam muita atenção na forma como usam computadores e softwares. Na maioria das vezes eles têm pouco conhecimento de truques, como atalhos de teclado, pesquisa e substituição global ou automação de tarefas comuns através de scripts.

- **Tópicos Avançados** cobrem material não-essencial ou mais difícil. Alguns desses tópicos introduzem construções sintáticas alternativas que não são necessariamente avançadas do ponto de vista técnico. Em muitos casos, o livro usa uma construção particular, mas explica alternativas como *Tópicos Avançados*. Sinta-se livre para utilizar essas construções nos seus próprios programas se preferir. Mas, de acordo com minha experiência, vários estudantes são gratos à abordagem “mantenha a coisa simples”, porque isso reduz significativamente o número de decisões desnecessárias que eles têm de tomar.

- **Fatos Aleatórios** fornecem informações históricas e sociais sobre a computação – para atender os requisitos de “contexto histórico e social” das diretrizes curriculares da ACM[1] – bem como pequenas pílulas de revisão de tópicos avançados em ciência da computação. Muitos estudantes lerão por conta própria os *Fatos Aleatórios* enquanto fingem seguir a aula.

- O ícone ao lado e o símbolo ⊕ direcionam os estudantes ao conteúdo online disponível no site www.bookman.com.br.

[1] Association for Computer Machinery.

Agradecimentos

Sou muito grato ao meu editor, Dan Sayre, por desenvolver continuamente este livro e os sites que o acompanham. Muito obrigado a Lauren Sapira, Todd Eaton, Don Fowley, Lindsay Murdock, Lisa Wojcik, Lisa Gee e Bud Peters da John Wiley & Sons, e a equipe da Publishing Services pela ajuda nesse projeto. Um reconhecimento especialmente sincero e um muito obrigado para Cindy Johnson pelo trabalho árduo, julgamento preciso e incrível atenção aos detalhes. Agradeço também a Rick Giles pelo trabalho nos slides.

Sou muito grato às várias pessoas que revisaram esta edição e as anteriores do livro, pelas sugestões valiosas feitas e que chamaram minha atenção para um número embaraçoso de erros e omissões. Elas incluem:

Tim Andersen, *Boise State University*
Ivan Bajic, *San Diego State University*
Ted Bangay, *Sheridan Institute of Technology*
George Basham, *Franklin University*
Sambit Bhattacharya, *Fayetteville State University*
Joseph Bowbeer, *Vizrea Corporation*
Timothy A. Budd, *Oregon State University*
Robert Burton, *Brigham Young University*
Frank Butt, *IBM*
Jerry Cain, *Stanford University*
Adam Cannon, *Columbia University*
Nancy Chase, *Gonzaga University*
Archana Chidanandan, *Rose-Hulman Institute of Technology*
Vincent Cicirello, *The Richard Stockton College of New Jersey*
Deborah Coleman, *Rochester Institute of Technology*
Valentino Crespi, *California State University, Los Angeles*
Jim Cross, *Auburn University*
Russell Deaton, *University of Arkansas*
H. E. Dunsmore, *Purdue University*
Robert Duvall, *Duke University*
Henry A. Etlinger, *Rochester Institute of Technology*
John Fendrich, *Bradley University*
John Fulton, *Franklin University*
David Geary, *Sabreware, Inc.*
Margaret Geroch, *Wheeling Jesuit University*
Rick Giles, *Acadia University*
Stacey Grasso, *College of San Mateo*
Jianchao Han, *California State University, Dominguez Hills*
Lisa Hansen, *Western New England College*
Elliotte Harold
Eileen Head, *Binghamton University*
Cecily Heiner, *University of Utah*
Brian Howard, *Depauw University*
Lubomir Ivanov, *Iona College*

Norman Jacobson, *University of California, Irvine*
Curt Jones, *Bloomsburg University*
Aaron Keen, *California Polytechnic State University, San Luis Obispo*
Elliot Koffman, *Temple University*
Kathy Liszka, *University of Akron*
Hunter Lloyd, *Montana State University*
Youmin Lu, *Bloomsburg University*
John S. Mallozzi, *Iona College*
John Martin, *North Dakota State University*
Scott McElfresh, *Carnegie Mellon University*
Joan McGrory, *Christian Brothers University*
Carolyn Miller, *North Carolina State University*
Teng Moh, *San Jose State University*
John Moore, *The Citadel*
Faye Navabi, *Arizona State University*
Kevin O'Gorman, *California Polytechnic State University, San Luis Obispo*
Michael Olan, *Richard Stockton College*
Kevin Parker, *Idaho State University*
Cornel Pokorny, *California Polytechnic State University, San Luis Obispo*
Roger Priebe, *University of Texas, Austin*
C. Robert Putnam, *California State University, Northridge*
Cyndi Rader, *Colorado School of Mines*
Neil Rankin, *Worcester Polytechnic Institute*
Brad Rippe, *Fullerton College*
Pedro I. Rivera Vega, *University of Puerto Rico, Mayaguez*
Daniel Rogers, *SUNY Brockport*
Carolyn Schauble, *Colorado State University*
Christian Shin, *SUNY Geneseo*
Jeffrey Six, *University of Delaware*
Don Slater, *Carnegie Mellon University*
Ken Slonneger, *University of Iowa*
Peter Stanchev, *Kettering University*
Ron Taylor, *Wright State University*
Joseph Vybihal, *McGill University*
Xiaoming Wei, *Iona College*
Todd Whittaker, *Franklin University*
Robert Willhoft, *Roberts Wesleyan College*
David Womack, *University of Texas at San Antonio*
Catherine Wyman, *DeVry University*
Arthur Yanushka, *Christian Brothers University*
Salih Yurttas, *Texas A&M University*

Sumário

Capítulo 1 Introdução 29

1.1 O que é programação? 30
1.2 A anatomia de um computador 31
1.3 Traduzindo programas legíveis pelo homem em código de máquina 36
1.4 A linguagem de programação Java 37
1.5 Conhecendo seu computador 40
1.6 Compilando um programa simples 43
1.7 Erros 49
1.8 O processo de compilação 51

Capítulo 2 Utilizando Objetos 59

2.1 Tipos e variáveis 60
2.2 O operador de atribuição 62
2.3 Objetos, classes e métodos 63
2.4 Parâmetros de método e valores de retorno 65
2.5 Tipos numéricos 68
2.6 Construindo objetos 69
2.7 Métodos de acesso e métodos modificadores 71
2.8T Implementando um programa de teste 72
2.9 A documentação da API 75
2.10 Referências a objetos 77
2.11G Aplicações gráficas e janelas de frame 80
2.12G Desenhando em um componente 82
2.13G Elipses, linhas, texto e cores 85

Capítulo 3 Implementando Classes 99

3.1 Níveis de abstração 100
3.2 Especificando a interface pública de uma classe 103
3.3 Comentando a interface pública 107
3.4 Campos de instância 110
3.5 Implementando construtores e métodos 112
3.6T Teste de unidade 119
3.7 Categorias de variáveis 121
3.8 Parâmetros de método implícitos e explícitos 124
3.9G Classes de formas gráficas 126

Capítulo 4 Tipos de Dados Fundamentais 145

4.1 Tipos numéricos 146
4.2 Constantes 149
4.3 Atribuição, incremento e decremento 155
4.4 Operações aritméticas e funções matemáticas 156
4.5 Chamando métodos estáticos 161
4.6 Strings 166
4.7 Lendo a entrada 169

Capítulo 5 Decisões 185

5.1 O comando `if` 186
5.2 Comparando valores 191
5.3 Múltiplas alternativas 196
5.4 Utilizando expressões booleanas 205
5.5T Cobertura do teste 210

Capítulo 6 Iteração 225

6.1 Laços `while` 226
6.2 Laços `for` 231
6.3 Laços aninhados 237
6.4 Processando valores sentinela 238
6.5 Números aleatórios e simulações 245
6.6T Utilizando um depurador 248
6.7T Uma sessão de depuração de exemplo 251

Capítulo 7 Arrays e Listas de Arrays 271

7.1 Arrays 272
7.2 Listas de arrays 276
7.3 Empacotadores e auto-empacotamento 281
7.4 O laço `for` aprimorado 283
7.5 Algoritmos simples para arrays 285
7.6 Arrays bidimensionais 290
7.7 Copiando arrays 294
7.8T Testes de regressão 300

Capítulo 8 Projetando Classes 315

8.1 Escolhendo classes 316
8.2 Coesão e acoplamento 318
8.3 Classes modificadoras, classes de acesso e classes imutáveis 321
8.4 Efeitos colaterais 321
8.5 Pré-condições e pós-condições 326
8.6 Métodos estáticos 329
8.7 Campos estáticos 331
8.8 Escopo 334
8.9 Pacotes 339
8.10T Frameworks de teste de unidade 345

Capítulo 9 Interfaces e Polimorfismo 359

9.1 Utilizando interfaces para reutilização de código 360
9.2 Convertendo entre classes e tipos interface 367
9.3 Polimorfismo 368
9.4 Utilizando interfaces para retornos de chamada 370
9.5 Classes internas 375
9.6G Eventos, fontes de eventos e ouvintes de eventos 377
9.7G Utilizando classes internas para ouvintes 381
9.8G Construindo aplicações com botões 384
9.9G Processando eventos de temporizador ⊕
9.10G Eventos de mouse ⊕

Capítulo 10 Herança 399

10.1 Uma introdução à herança 400
10.2 Hierarquias de heranças 405
10.3 Herdando campos e métodos de instância 407
10.4 Construção das subclasses 413
10.5 Convertendo subclasses em superclasses 414
10.6 Polimorfismo 417
10.7 Controle de acesso 422
10.8 `Object`: a superclasse cósmica 425
10.9G Utilizando herança para personalizar frames 432
10.10G Processando entrada de texto 434
10.11G Áreas de texto 437

Capítulo 11 Entrada/Saída e Tratamento de Exceção 451

11.1 Lendo e gravando arquivos de texto 452
11.2 Lançando exceções 455
11.3 Exceções verificadas e não-verificadas 458
11.4 Capturando exceções 460
11.5 A cláusula `finally` 463
11.6 Projetando seus próprios tipos de exceção 465
11.7 Estudo de caso: um exemplo completo 467

Capítulo 12 Projeto Orientado a Objetos 479

12.1 O ciclo de vida de um software 480
12.2 Descobrindo classes 485
12.3 Relacionamentos entre classes 488
12.4 Estudo de caso: Imprimindo uma fatura 491
12.5 Estudo de caso: Um caixa automático 503

Capítulo 13 Recursão 533

13.1 Números triangulares 534
13.2 Permutações 538
13.3 Métodos auxiliares recursivos 546
13.4 A eficiência da recursão 548
13.5 Recursões mútuas 554

Capítulo 14 Classificação e Pesquisa 571

14.1 Classificação por seleção 572
14.2 Medindo o tempo de execução do algoritmo de classificação por seleção 575
14.3 Analisando o desempenho do algoritmo de classificação por seleção 579
14.4 Classificação por intercalação 581
14.5 Analisando o algoritmo de classificação por intercalação 584
14.6 Pesquisa 588
14.7 Pesquisa binária 590
14.8 Classificando dados reais 593

Capítulo 15 Uma Introdução a Estruturas de Dados 603

15.1 Utilizando listas encadeadas 604
15.2 Implementando listas encadeadas 609
15.3 Tipos abstratos de dados e tipos concretos de dados 619
15.4 Pilhas e filas 623

Capítulo 16 Estruturas de Dados Avançadas (avançado)

16.1 Conjuntos
16.2 Mapas
16.3 Tabelas de hash
16.4 Calculando códigos de hash
16.5 Árvores binárias de pesquisa
16.6 Percurso em árvore
16.7 Utilizando conjuntos de árvores e mapas de árvores
16.8 Filas de prioridade
16.9 Heaps
16.10 O algoritmo heapsort

Capítulo 17 Programação Genérica (avançado)

17.1 Variáveis de tipo
17.2 Implementando classes genéricas
17.3 Métodos genéricos
17.4 Restringindo variáveis de tipo
17.5 Tipos brutos

Capítulo 18 Interfaces Gráficas com o Usuário (avançado) ⊕

18.1G Gerenciamento de leiaute
18.2G Opções
18.3G Menus
18.4G Explorando a documentação do Swing

Apêndice A Guia de Estilo para Codificação em Java 635

Apêndice B Os Subconjuntos Basic Latin e Latin-1 do Unicode 643

Apêndice C A Biblioteca Java 646

Apêndice D Resumo de Sintaxe Java ⊕

Apêndice E Resumo de Operadores Java ⊕

Apêndice F Resumo de Palavras-chave em Java ⊕

Apêndice G Fatores da Conversão Métrica ⊕

Apêndice H Resumo sobre HTML ⊕

Apêndice I Resumo das Ferramentas ⊕

Apêndice J Resumo de javadoc ⊕

Apêndice K Sistemas Numéricos ⊕

Apêndice L Operações de Bits e Deslocamentos ⊕

Apêndice M Resumo de UML ⊕

Glossário 681

Índice 695

Créditos das Ilustrações 719

Lista das caixas de sintaxe em ordem alfabética

A instrução `return` **114**
Acesso ao elemento do array **275**
Assertiva **327**
Atribuição **63**
Bloco de instruções **189**
Bloco `try` geral **461**
Chamada de método **48**
Chamada de método estático **162**
Chamando um construtor da superclasse **413**
Chamando um método de superclasse **411**
Classes internas **375**
Cláusula `finally` **464**
Coerção (Typecasting) **148**
Construção de array **274**
Construção de objeto **71**
Declaração de campo de instância **112**
Definição de classe **107**
Definição de constante **151**
Definição de construtor **106**
Definição de método **105**
Definição de variável **61**
Definindo um método genérico ⊕
Definindo um tipo enumerado ⊕
Definindo uma classe genérica ⊕
Definindo uma interface **364**
Especificação de exceção **460**
Especificação de pacote **340**
Herança **403**
Implementando uma interface **364**
Importando uma classe a partir de um pacote **74**
Instanciando uma classe genérica ⊕
Lançando uma exceção **457**
O comando `if` **188**
O comando instrução `for` **231**
O comando instrução `while` **228**
O laço "for each" **285**
O operador `instanceof` **416**

Capítulo	Erros Comuns	Como Fazer	Dicas de Qualidade
1 Introdução	Omitindo ponto-e-vírgulas 48 Palavras digitadas incorretamente 51		
2 Utilizando objetos	Tentando invocar um construtor como se fosse um método 71		
3 Implementando classes	Esquecendo de inicializar referências a objetos em um construtor 123 Tentando chamar um método sem um parâmetro implícito 125	Implementando uma classe 115 Desenhando formas gráficas 131	
4 Tipos de dados fundamentais	Divisão de inteiros 159 Parênteses desbalanceados 160 Erros de arredondamento 163	Realizando cálculos 163	Não utilize números mágicos 154 Escolha nomes descritivos para variáveis 154 Espaço em branco 160 Fatorando código comum 161
5 Decisões	Utilizando `==` para comparar strings 194 O problema do `else` oscilante 203 Múltiplos operadores relacionais 207 Confundindo condições `&&` com `||` 208		Leiaute de chaves 189 Evite condições com efeitos colaterais 196 Calcule os dados de exemplo manualmente 211 Prepare os casos de teste antecipadamente 212
6 Iteração	Laços infinitos 229 Erros "por um" 229 Esquecendo um ponto-e-vírgula 236 Um ponto-e-vírgula em excesso 236	Implementando laços 242 Depurando 257	Utilize o laço `for` apenas para os propósitos para os quais ele foi projetado 235 Não utilize `!=` para testar o fim de um intervalo 237 Limites simétricos e assimétricos 244 Conte as iterações 244

Dicas de Produtividade	Tópicos Avançados	Fatos Aleatórios
Entenda o sistema de arquivos 42 Tenha uma estratégia de backup ⊕	Sintaxe alternativa de comentário 49	O ENIAC e a alvorada da computação ⊕
Não memorize – use a ajuda on-line 77	Testando classes em um ambiente interativo ⊕ Applets ⊕	Mainframes – quando os dinossauros dominavam a Terra ⊕ A evolução da Internet ⊕
O utilitário `javadoc` ⊕ Utilizando a linha de comando de maneira eficiente ⊕	Chamando um construtor a partir de um outro ⊕	Urnas eletrônicas ⊕ Computação gráfica ⊕
Evite um leiaute instável ⊕ Lendo relatórios de exceções 168	Números grandes ⊕ Números binários ⊕ Combinando atribuição e aritmética ⊕ Seqüências de escape ⊕ Strings e o tipo `char` ⊕ Formatando números ⊕ Utilizando caixas de diálogo para entrada e saída ⊕	O bug de ponto flutuante do Pentium ⊕ Alfabetos internacionais ⊕
Recuos e tabulações 190 Atalhos de teclado para operações do mouse ⊕ Copie e cole no editor ⊕ Crie um cronograma e reserve tempo para problemas inesperados 204	Operador de seleção ⊕ O comando `switch` ⊕ Tipos enumerados ⊕ Avaliação preguiçosa dos operadores booleanos ⊕ Lei de De Morgan ⊕ Registro em log 212	Inteligência artificial ⊕
	Laços do ⊕ Variáveis definidas no cabeçalho de um laço `for` ⊕ O problema do "um laço e meio" ⊕ Os comandos `break` e `continue` ⊕ Invariantes de laço ⊕	Código espaguete ⊕ Provas de correção ⊕ O primeiro bug ⊕

Capítulo	Erros Comuns	Como Fazer	Dicas de Qualidade
7 Arrays e listas de arrays	Erros de limite 275 Arrays não-inicializados 276 Comprimento e tamanho 281 Subestimando o tamanho de um conjunto de dados 298	Trabalhando com arrays e listas de arrays 293	Prefira listas de arrays parametrizadas ⊕ Transforme arrays paralelos em arrays de objetos 298
8 Projetando classes	Tentando modificar parâmetros de tipos primitivos 323 Sombreamento 337 Uso confuso de pontos 343	Programando com pacotes 346	Consistência 320 Minimize os efeitos colaterais 325 Não altere o conteúdo das variáveis de parâmetros 325
9 Interfaces e polimorfismo	Esquecendo de definir a implementação de métodos como públicos 366 Tentando instanciar uma interface 368 Modificando a assinatura na implementação do método 380 Esquecendo de relacionar um ouvinte 387 Esquecendo de repintar 388		
10 Herança	Confundindo superclasses e subclasses 404 Sombreando campos de instância 411 Falhando ao invocar o método da superclasse 412 Acesso acidental a pacote 424 Tornando métodos herdados menos acessíveis 424 Definindo o método equa1s com o tipo errado de parâmetro 429 Esquecendo de clonar 430 Por padrão, componentes têm largura e altura zero 441	Implementando uma interface gráfica com o usuário (*graphical user interface* – GUI) 439	Clone campos de instância mutáveis em métodos de acesso ⊕

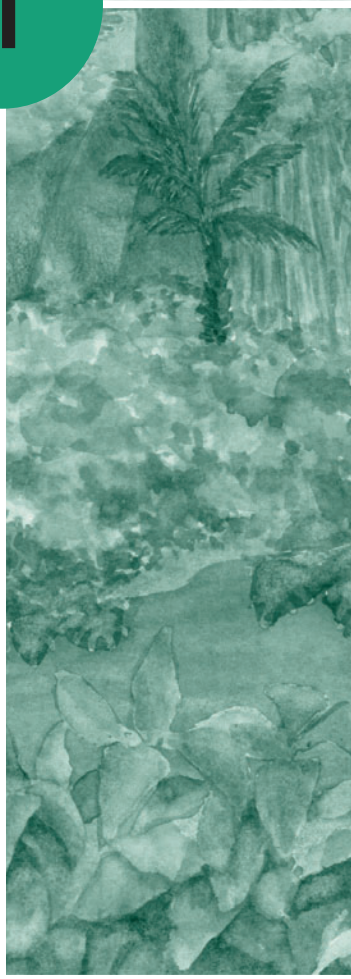
Dicas de Produtividade	Tópicos Avançados	Fatos Aleatórios
Arquivos batch e scripts de shell ⊕	Inicialização de array 276 Arrays bidimensionais com comprimentos de linha variáveis ⊕ Arrays multidimensionais ⊕ Arrays parcialmente preenchidos ⊕ Métodos com um número variável de parâmetros ⊕	Um worm no início da Internet ⊕ Os incidentes do Therac-25 ⊕
Pesquisa e substituição global ⊕ Expressões regulares ⊕	Chamada por valor e chamada por referência ⊕ Invariantes de classe ⊕ Formas alternativas de inicialização de campos ⊕ Importações estáticas ⊕	O crescimento explosivo dos computadores pessoais ⊕
Não utilize um contêiner como um ouvinte 387	Constantes em interfaces 366 Classes anônimas ⊕ Adaptadores de evento ⊕	Sistemas operacionais ⊕ Linguagens de programação ⊕
Fornecer `toString` em todas as classes 427 Reutilização de código 441	Classes abstratas ⊕ Métodos e classes final ⊕ Acesso protegido ⊕ Herança e o método `toString` ⊕ Herança e o método `equals` ⊕ Implementando o método `clone` ⊕ Tipos enumerados revistos ⊕ Adicionando o método `main` à classe de frame 433	Linguagens de criação de scripts ⊕

Capítulo	Erros Comuns	Como Fazer	Dicas de Qualidade
11 Entrada/saída e tratamento de exceção	Barras invertidas em nomes de arquivo 454		Lance cedo, capture tarde 462 Não silencie exceções 463 Não utilize `catch` e `finally` no mesmo comando `try` 465 Lance exceções específicas 466
12 Projeto orientado a objetos		Cartões CRC e diagramas UML 490	
13 Recursão	Recursão infinita 538 Rastreando métodos recursivos 542	Pensando recursivamente 543	
14 Classificação e pesquisa	O método `compareTo` pode retornar qualquer inteiro, não só –1, 0 e 1 595		
15 Uma introdução a estruturas de dados			
16 Estruturas de dados avançadas (avançado)	Esquecendo de definir `hashCode` ⊕	Escolhendo um contêiner ⊕	Utilize referências de interface para manipular estruturas de dados ⊕
17 Programação genérica (avançado)	Genericidade e herança ⊕ Escrevendo código que não funciona depois que os tipos são apagados ⊕ Utilizando tipos genéricos em um contexto estático ⊕		
18 Interfaces gráficas com o usuário (avançado)		Criando uma interface com o usuário ⊕	

Dicas de Produtividade	Tópicos Avançados	Fatos Aleatórios
	Caixas de diálogo de arquivo Argumentos de linha de comando	O incidente com o foguete Ariane
	Atributos e métodos em diagramas UML Multiplicidades Agregação e associação	Produtividade do programador Desenvolvimento de software – arte ou ciência?
		Os limites da computação
	Classificação por inserção O, Ômega e Teta O algoritmo quicksort A interface `Comparable` parametrizada 595 A interface `Comparator`	A primeira programadora
	A interface `Iterable` e o laço "For Each" Classes internas estáticas	Padronização
		Notação polonesa invertida Pirataria de software
	Tipos curinga	
Utilize um construtor de interfaces com o usuário		

Capítulo 1

Introdução



Objetivos do capítulo

- Entender a atividade de programação
- Aprender sobre arquitetura e computadores
- Aprender sobre código de máquina e linguagens de programação de alto nível
- Familiarizar-se com seu ambiente de computação e seu compilador
- Compilar e executar seu primeiro programa em Java
- Reconhecer erros de sintaxe e de lógica

O objetivo deste capítulo é familiarizá-lo com o conceito de programação. Ele revisa a arquitetura de um computador e discute a diferença entre código de máquina e as linguagens de programação de alto nível. Por fim, você verá como compilar e executar seu primeiro programa em Java e como diagnosticar erros que podem ocorrer quando um programa é compilado ou executado.

Conteúdo do Capítulo

1.1 O que é programação? 30

1.2 A anatomia de um computador 31

Fato aleatório 1.1: O ENIAC e a alvorada da computação

1.3 Traduzindo programas legíveis pelo homem em código de máquina 36

1.4 A linguagem de programação Java 37

1.5 Conhecendo seu computador 40

Dica de produtividade 1.1: Entenda o sistema de arquivos 42

Dica de produtividade 1.2: Tenha uma estratégia de backup

1.6 Compilando um programa simples 43

Sintaxe 1.1: Chamada de método 48

Erro comum 1.1: Omitindo ponto-e-vírgulas 48

Tópico avançado 1.1: Sintaxe alternativa de comentário 49

1.7 Erros 49

Erro comum 1.2: Palavras digitadas incorretamente 51

1.8 O processo de compilação 51

1.1 O que é programação?

Provavelmente você já utilizou um computador para trabalho ou lazer. Muitas pessoas utilizam computadores para tarefas cotidianas como consultar o saldo bancário ou escrever um trabalho escolar. Computadores são bons para essas tarefas. Eles podem tratar tarefas repetitivas, como somar números ou inserir palavras em uma página, sem ficar entediados ou exaustos. Os computadores também são bons para jogos porque podem reproduzir seqüências de sons e imagens, envolvendo o usuário humano no processo.

A flexibilidade de um computador é um fenômeno bem surpreendente. A mesma máquina pode ser usada para consultar um saldo bancário, imprimir um trabalho escolar e jogar um jogo. Em comparação, outros tipos de máquinas executam uma variedade bem menor de tarefas – um carro anda e uma torradeira torra.

Um computador precisa ser programado para realizar tarefas. Diferentes tarefas requerem diferentes programas.

Um programa de computador executa uma seqüência de operações muito básicas em rápida sucessão.

Para alcançar essa flexibilidade, o computador deve ser *programado* para realizar cada tarefa. O computador é uma máquina que armazena dados (números, palavras, imagens), interage com dispositivos (monitor, sistema de som, impressora) e executa programas. Programas são seqüências de instruções e decisões que o computador executa para realizar uma tarefa. Um programa calcula o saldo bancário; um outro, talvez projetado e construído por uma empresa diferente, processa texto; e um terceiro programa, provavelmente de outra empresa, executa um jogo.

Os programas dos computadores atuais são tão sofisticados que é difícil acreditar que são todos compostos a partir de operações extremamente primitivas.

Uma operação típica pode ser uma das listadas abaixo:

- Colocar um ponto vermelho nesta posição da tela.
- Enviar a letra A para a impressora.
- Obter um número a partir desta posição de memória.
- Somar dois números.
- Se este valor for negativo, continuar o programa a partir desta instrução.

Um programa de computador contém as seqüências de instruções para todas as tarefas que pode executar.

Um programa informa para um computador, em detalhes, a seqüência dos passos necessários para completar uma tarefa. Um programa contém um número enorme de operações simples e o computador as executa em grande velocidade. O computador não possui inteligência – ele simplesmente executa seqüências de instruções que foram previamente preparadas.

Para utilizar um computador, não é necessário ter conhecimento sobre programação. Quando você escreve um trabalho escolar usando um processador de texto, esse pacote de software foi programado pelo fabricante e está pronto para seu uso. Isso é apenas o esperado – você pode dirigir um carro sem ser mecânico e tostar uma fatia de pão sem ser eletricista.

O principal propósito deste livro é ensinar como projetar e implementar programas de computador. Você aprenderá a formular instruções para todas as tarefas que seus programas precisam executar.

Tenha em mente que programar um jogo de computador sofisticado ou um processador de texto requer uma equipe de muitos programadores altamente qualificados, artistas gráficos e outros profissionais. Seus primeiros esforços de programação serão mais triviais. Os conceitos e as habilidades que você aprenderá neste livro formam uma base importante, mas não espere produzir imediatamente softwares profissionais. Um curso universitário em ciência de computação ou engenharia de software típico leva quatro anos para ser concluído. O objetivo deste livro é ser um curso introdutório em um curso universitário completo.

Muitos alunos acham que há uma grande emoção mesmo nas tarefas mais simples de programação. É uma experiência incrível ver o computador executar de maneira tão precisa e rápida uma tarefa que demandaria horas de trabalho árduo.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. O que é necessário para tocar um CD de músicas em um computador?
2. Por que um CD player é menos flexível que um computador?
3. Um programa de computador pode tomar a iniciativa de executar tarefas de uma maneira melhor do que seus programadores imaginaram?

1.2 A anatomia de um computador

Para entender o processo de programação, você precisa ter pelo menos um entendimento rudimentar dos blocos de construção que compõem um computador. Esta seção descreverá um computador pessoal. Computadores maiores têm componentes maiores, mais poderosos ou mais rápidos, mas fundamentalmente todos possuem as mesmas partes.

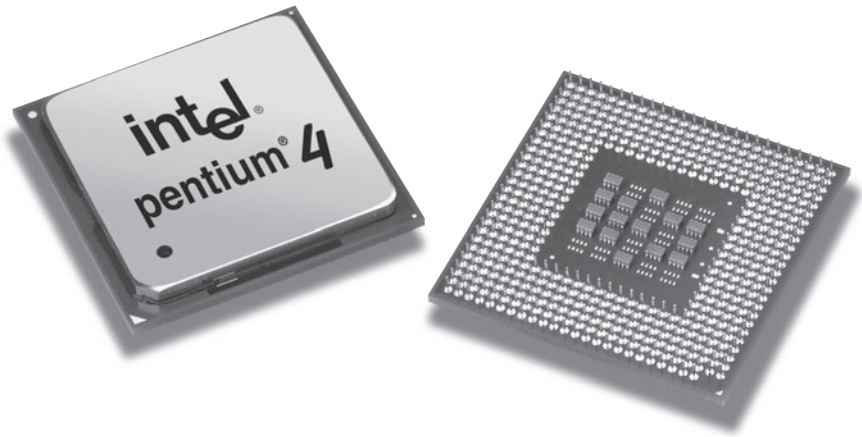


Figura 1 Unidade central de processamento.

No coração do computador está a CPU (unidade central de processamento).

No coração do computador está a *unidade central de processamento* – UCP (*central processing unit* – CPU*) (veja Figura 1). Ela consiste em um único *chip* (circuito integrado) ou em um pequeno número de chips. Um chip de computador é um componente com uma embalagem plástica ou metálica, com conectores de metal e fiação interna feita principalmente de silício. Em um chip de CPU, a fiação interna é muito complicada. Por exemplo, o chip do Pentium 4 (uma CPU popular em computadores pessoais no momento em que escrevíamos este livro) contém mais de 50 milhões de elementos estruturais chamados *transistores* – elementos que permitem que sinais elétricos controlem outros sinais elétricos, tornando possível a computação automática. A CPU localiza e executa instruções de programa; executa operações aritméticas como adição, subtração, multiplicação e divisão; busca dados nas unidades de armazenamento e dispositivos de entrada/saída e envia esses dados de volta.

Dados e programas são mantidos na unidade de armazenamento primária (memória) e nas unidades de armazenamento secundárias (por exemplo, um disco rígido).

O computador mantém os dados e os programas em unidades de *armazenamento*. Há dois tipos de armazenamento. O *armazenamento primário*, também chamado *memória de acesso aleatório* (*random-access memory* – RAM) ou simplesmente *memória*, é rápido, porém, caro e consiste em chips de memória (veja Figura 2). O armazenamento primário tem duas desvantagens: é comparativamente caro e perde todos os dados quando o computador é desligado. O *armazenamento secundário*, normalmente um *disco rígido* (veja Figura 3), oferece um armazenamento mais barato que persiste sem eletricidade. Um disco rígido consiste em lâminas giratórias revestidas por um material magnético e em cabeças de leitura/gravação que podem detectar e alterar os padrões de variação no fluxo magnético sobre as lâminas. Basicamente, é o mesmo processo de gravação e reprodução utilizado em fitas de áudio ou vídeo.

* N. de R.T.: Adotaremos o acrônimo em inglês, CPU, por ser bastante popular no Brasil.

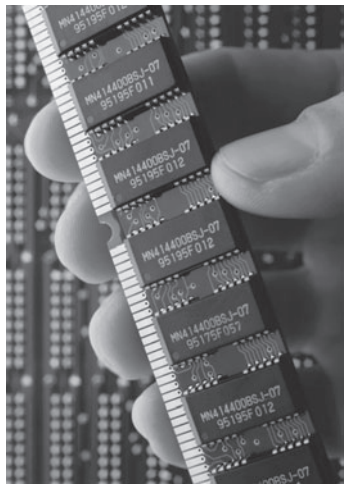


Figura 2 Módulo de memória com chips de memória.

Alguns computadores são unidades autocontidas, enquanto outros são interconectados em *redes*. Na maioria das vezes, os computadores domésticos permanecem intermitentemente conectados à Internet via conexão discada ou de banda larga. Computadores em um laboratório de computação provavelmente permanecem conectados a uma rede local. Por meio do cabeamento de rede, o computador pode ler programas armazenados

Figura 3 Disco rígido.

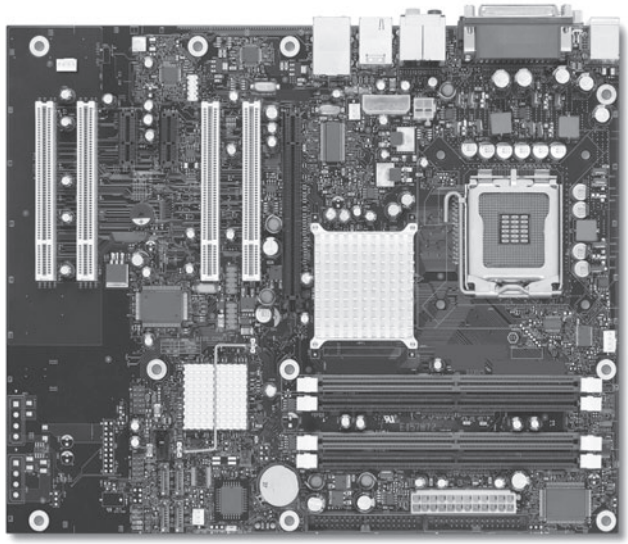


Figura 4 Placa-mãe.

em unidades de armazenamento centralizadas ou enviar dados a outros computadores. Para o usuário de um computador em rede, pode não ser óbvio distinguir quais dados residem no próprio computador e quais são transmitidos pela rede.

A maioria dos computadores tem dispositivos de *armazenamento removíveis* que permitem acessar dados ou programas em mídias como disquetes, fitas ou discos compactos (*compact discs* – CDs).

Para interagir com um usuário humano, um computador requer outros dispositivos periféricos. O computador transmite as informações ao usuário através de uma tela de vídeo, alto-falantes e impressoras. O usuário pode inserir informações e instruções para o computador utilizando um teclado ou um dispositivo de apontamento como um mouse.

A CPU, a RAM e os circuitos eletrônicos que controlam o disco rígido e outros dispositivos são interconectados por um conjunto de linhas elétricas chamado *barramento*. Os dados trafegam através do barramento indo e voltando entre a memória do sistema ou os dispositivos periféricos e a CPU. A Figura 4 mostra uma *placa-mãe*, que contém a CPU, a RAM e conectores para dispositivos periféricos.

A CPU lê as instruções de máquina a partir da memória. As instruções a orientam a se comunicar com a memória, os dispositivos de armazenamento secundário e os periféricos.

A Figura 5 fornece uma visão esquemática geral da arquitetura de um computador. As instruções e os dados de um programa (como texto, números, áudio ou vídeo) são armazenados no disco rígido, em um CD ou estão disponíveis pela rede. Quando um programa é iniciado, ele é carregado na memória onde pode ser lido pela CPU. A CPU lê o programa uma instrução por vez. Guiada por essas instruções, a CPU lê os dados, modifica-os e grava-os de volta na RAM ou no dispositivo de armazenamento secundário. Algumas instruções de programa farão a CPU inte-

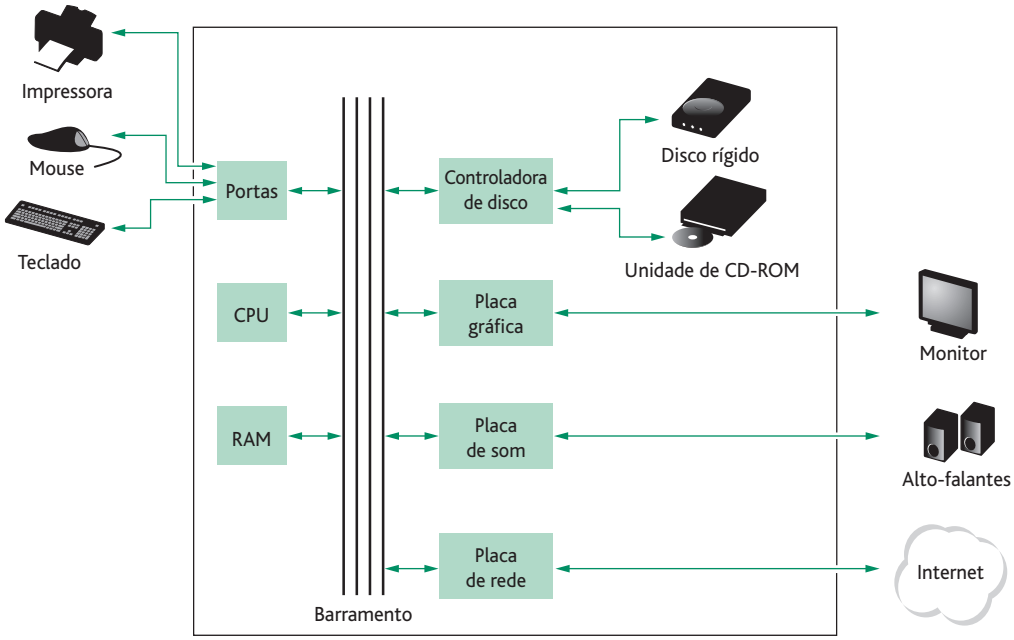


Figura 5 Diagrama esquemático de um computador.

ragir com os dispositivos que controlam a tela de vídeo ou o alto-falante. Como essas ações acontecem muitas vezes e em grande velocidade, o usuário perceberá imagens e som. De maneira semelhante, a CPU pode enviar instruções a uma impressora para marcar o papel com padrões de pontos tão pequenos e próximos que a visão humana reconhece como caracteres de texto e imagens. Algumas instruções de programa lêem a entrada do usuário a partir do teclado ou do mouse. O programa analisa a natureza dessas entradas e então executa as próximas instruções apropriadas.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

4. Onde um programa é armazenado quando não está em execução?
5. Qual parte do computador realiza operações aritméticas, como adição e multiplicação?

FATO ALEATÓRIO 1.1

O ENIAC e a alvorada da computação

O Fato Aleatório 1.1 comenta a história do ENIAC, o primeiro computador eletrônico utilizável. O ENIAC, concluído em 1946, continha aproximadamente 18.000 válvulas a vácuo e ocupava uma sala grande inteira.

1.3 Traduzindo programas legíveis pelo homem em código de máquina

Geralmente, o código de máquina depende do tipo de CPU. Entretanto, o conjunto de instruções da máquina virtual Java (JVM – Java Virtual Machine) pode ser executado em muitas CPUs.

No nível mais básico, instruções de computador são extremamente primitivas. O processador executa *instruções de máquina.* As CPUs de diferentes fornecedores, como o Pentium da Intel ou a SPARC da Sun, têm conjuntos de instruções de máquina diferentes. Para que aplicativos Java possam executar em múltiplas CPUs sem modificação, programas Java contêm instruções de máquina para uma "Máquina Virtual Java" (JVM ou Java Virtual Machine), uma CPU idealizada e que é simulada pela execução de um programa na CPU real. A diferença entre as instruções de máquina virtual e de máquina real não é importante – tudo o que você precisa saber é que instruções de máquina são muito simples; são codificadas como números, armazenadas na memória e podem ser executadas muito rapidamente.

Uma seqüência típica de instruções de máquina é

1. Carregue o conteúdo armazenado na posição 40 da memória.
2. Carregue o valor 100.
3. Se o primeiro valor for maior que o segundo, continue com a instrução que está armazenada na posição 240 da memória.

Na verdade, instruções de máquina são codificadas como números para que possam ser armazenadas na memória. Na máquina virtual Java, essa seqüência de instruções é codificada como a seqüência de números

```
21 40
16 100
163 240
```

Quando a máquina virtual busca essa seqüência de números, ela a decodifica e executa a seqüência de comandos associada.

Uma vez que as instruções de máquina são codificadas como números, é difícil escrever programas em código de máquina.

Como você pode fornecer a seqüência de comando para o computador? O método mais direto é colocar os próprios números na memória do computador. Essa é, na realidade, a maneira como os primeiros computadores funcionavam. Mas um programa longo é composto de milhares de comandos individuais e é entediante e suscetível a erros pesquisar os códigos numéricos para todos os comandos e colocá-los manualmente na memória. Como dissemos anteriormente, computadores são realmente bons para automatizar atividades entediantes e sujeitas a erros, e não demorou muito para que os programadores percebessem que os computadores poderiam ajudar no próprio processo de programação.

Linguagens de alto nível permitem descrever tarefas em um nível conceitual mais alto que o do código de máquina.

Na metade dos anos 1950, começaram a aparecer linguagens de programação *de alto nível*. Nessas linguagens, o programador expressa a idéia por trás da tarefa que precisa ser realizada e um programa de computador especial, chamado *compilador*, traduz a descrição de alto nível em instruções de máquina para um processador particular.

Por exemplo, em Java, a linguagem de programação de alto nível utilizada neste livro, você poderia escrever a seguinte instrução:

```
if (intRate > 100)
   System.out.println("Interest rate error");
```

Isso significa "Se a taxa de juros estiver acima de 100, exiba uma mensagem de erro". É tarefa do programa compilador examinar a seqüência de caracteres `if (intRate > 100)` e traduzi-la em:

```
21 40 16 100 163 240 . . .
```

Um compilador traduz programas escritos em uma linguagem de alto nível para código de máquina.

Compiladores são programas bem sofisticados. Eles traduzem instruções lógicas, como a instrução `if`, em seqüências de cálculos, testes e saltos. Eles atribuem posições de memória a *variáveis* – partes de informação identificadas por nomes simbólicos – como `intRate`. Neste curso, a existência de um compilador estará subentendida na maioria das vezes. Se decidir tornar-se um cientista da computação profissional, você irá aprender mais sobre como escrever programas compiladores mais adiante em seus estudos.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

6. O que é o código para a instrução da máquina virtual Java "Carregue o conteúdo armazenado na posição 100 da memória"?
7. Uma pessoa que usa um computador para trabalho administrativo precisará executar um compilador?

1.4 A linguagem de programação Java

Java foi originalmente projetado para a programação de dispositivos de consumo popular, mas inicialmente foi utilizado com sucesso para escrever *applets* para Internet.

Em 1991, um grupo liderado por James Gosling e Patrick Naughton, da Sun Microsystems, projetou uma linguagem de programação denominada "Green" para uso em dispositivos de consumo popular, como "set-top boxes" de televisores inteligentes. Essa linguagem foi projetada para ser simples e de arquitetura neutra para que pudesse ser executada em vários tipos de hardware. Não havia clientes para essa tecnologia.

Gosling relembra que, em 1994, a equipe percebeu que "Poderíamos escrever um navegador realmente legal. Era uma das poucas coisas na tendência dominante cliente/servidor que precisava de algumas das esquisitisses que fizemos: independência de arquitetura, com processamento em tempo real, confiável e seguro". Java foi apresentado a uma multidão entusiasmada na exposição SunWorld em 1995.

Java foi projetado para ser seguro e portável, beneficiando tanto usuários da Internet como estudantes.

Desde então, Java tem crescido a uma velocidade fenomenal. Programadores escolheram a linguagem porque ela é mais simples do que sua concorente mais próxima, C++. Além disso, Java tem uma rica *biblioteca* que possibilita escrever programas portáveis capazes de ignorar os sistemas operacionais proprietários – um recurso avidamente perseguido por aqueles que queriam independência em relação a esses sistemas e que foi ferozmente combatido pelos fornecedores. Uma "micro edição" e uma "edição corporativa" da biblioteca Java fez os programadores Java se sentirem em casa

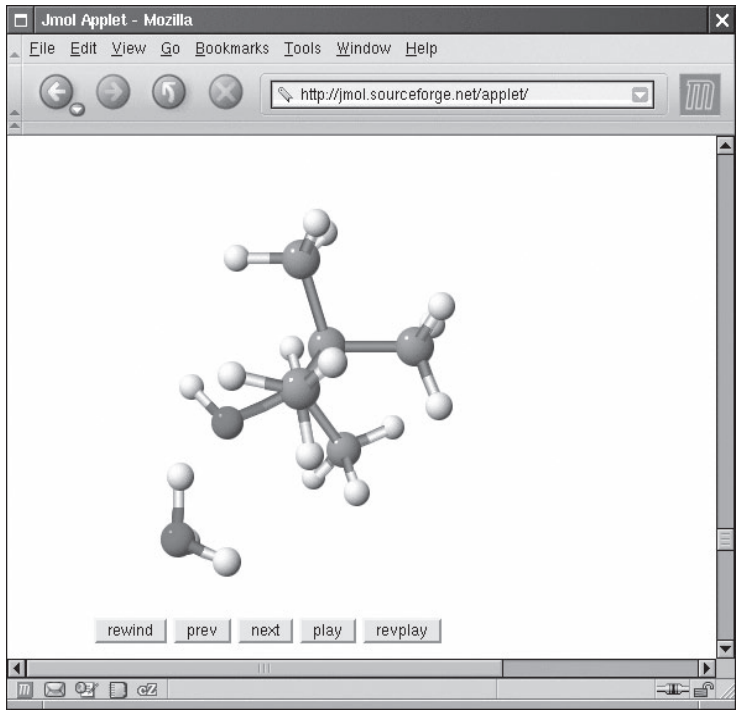


Figura 6 Applet para visualização de moléculas ([1]).

quanto a diferentes tipos de hardware, desde cartões inteligentes e telefones celulares até os maiores servidores de Internet.

Como a linguagem Java foi projetada para a Internet, ela tem dois atributos que a tornam bem adequada para iniciantes: segurança e portabilidade. Se visitar uma página Web que contém código Java (as chamadas *applets* – veja um exemplo na Figura 6), o código começa a executar automaticamente. É importante salientar que você pode confiar que applets são inerentemente seguras. Se uma applet pudesse fazer algo nocivo, como danificar dados ou ler informações pessoais no seu computador, você estaria sob perigo real toda vez que navegasse pela Web – um projetista inescrupuloso poderia disponibilizar uma página Web contendo código perigoso que seria executado na sua máquina assim que você visitasse a página. A linguagem Java tem vários recursos de segurança que garantem que nenhuma applet nociva possa ser executada no seu computador. Como um benefício extra, esses recursos também o ajudam a aprender a linguagem mais rapidamente. A máquina virtual Java pode capturar muitos tipos de erro de iniciantes e informá-los de uma maneira precisa. (Em comparação, muitos erros de iniciantes na linguagem C++ produzem programas que agem de maneira aleatória e confusa.) Outro benefício da linguagem Java é a portabilidade. O mesmo programa Java executará, sem modificação, em Windows, UNIX, Linux ou Macintosh. A portabilidade também é um requisito para applets. Quando você visita uma página Web, o servidor Web que disponibiliza o conteúdo da página não faz idéia do computador que você está utilizando para navegar pela Web. Ele simplesmente retorna o código portável que foi gerado pelo compilador Java. A máquina virtual no seu computador é quem

executa esse código portável. Novamente, há um benefício para o estudante. Você não precisa aprender a escrever programas para diferentes sistemas operacionais.

Atualmente, Java ocupa uma posição sólida como uma das linguagens mais importantes para programação de uso geral e também para ensino de ciência da computação. Mas, embora Java seja uma boa linguagem para iniciantes, ela não é perfeita por três razões.

Como Java não foi especificamente projetado para estudantes, nenhum esforço foi feito para torná-lo realmente simples para escrever programas básicos. Java exige certos requisitos de hardware para escrever até mesmo os programas mais simples. Isso não é um problema para programadores profissionais, mas é uma desvantagem para estudantes iniciantes. À medida que você aprende a programar em Java, haverá momentos em que você terá de se satisfazer com uma explicação preliminar e esperar os detalhes completos em um capítulo posterior.

A linguagem Java foi revisada e estendida várias vezes durante sua vida – veja a Tabela 1. Neste livro, supomos que você instalou a versão Java 5 ou superior.

Java tem uma biblioteca bastante extensa. Focalize o aprendizado naquelas partes da biblioteca que você precisa para seus projetos de programação.

Por fim, não espere aprender tudo sobre Java em um semestre. A linguagem Java em si é relativamente simples, mas contém um amplo conjunto de *pacotes de biblioteca* que são necessários para escrever programas úteis. Há pacotes para gráficos, projetos de interfaces com o usuário, criptografia, rede, som, armazenamento em bancos de dados e muitos outros propósitos. Mesmo programadores Java especialistas não podem esperar conhecer o conteúdo de todos os pacotes – eles utilizam apenas os pacotes necessários para projetos específicos.

Com este livro, você deve esperar aprender boa parte da linguagem Java e dos pacotes mais importantes. Tenha em mente que o objetivo central deste livro não é fazê-lo memorizar as minúcias do Java, mas ensiná-lo como pensar sobre programação.

Tabela 1 Versões de Java

Versão	Ano	Novos recursos importantes
1.0	1996	
1.1	1997	Classes internas
1.2	1998	Swing, coleções
1.3	2000	Melhorias de desempenho
1.4	2002	Assertivas, XML
5	2004	Classes genéricas, laço `for` melhorado, auto-empacotamento, enumerações
6	2006	Melhorias nas bibliotecas

AUTOVERIFICAÇÃO DA APRENDIZAGEM

8. Quais são os dois benefícios mais importantes da linguagem Java?
9. Quanto tempo leva para aprender toda a biblioteca Java?

1.5 Conhecendo seu computador

Reserve algum tempo para se familiarizar com o sistema computacional e o compilador Java que você utilizará para seus trabalhos escolares.

Talvez você esteja cursando sua primeira disciplina de programação ao ler este livro ou talvez esteja trabalhando em um sistema computacional com o qual esteja pouco familiarizado. Invista algum tempo conhecendo o computador. Como os sistemas computacionais variam muito, este livro só pode fornecer um esboço dos passos que você precisa seguir. Utilizar um sistema computacional novo e pouco conhecido pode ser frustrante, especialmente se você estiver sozinho. Procure treinamentos em sua universidade ou peça para um amigo lhe ensinar os princípios básicos do sistema.

Passo 1. Login

Se estiver utilizando seu computador pessoal, provavelmente não precisará se preocupar com esse passo. Mas computadores em um laboratório normalmente não estão abertos para todo mundo. Talvez você precise de um nome ou número de conta e uma senha para ter acesso a esse sistema.

Passo 2. Localize o compilador Java

Sistemas computacionais diferem muito quanto a isto. Em alguns sistemas você precisa abrir uma *janela de shell* (veja Figura 7) e digitar comandos para carregar o compilador. Outros sistemas têm um *ambiente de desenvolvimento integrado* no qual você pode escrever e testar seus programas (veja Figura 8). Muitos laboratórios de universidades possuem material de informação e tutoriais com orientações sobre as ferramentas instaladas no laboratório. Há instruções para vários compiladores populares disponíveis no WileyPLUS (recurso da editora digital).

Passo 3. Entenda arquivos e pastas

Como um programador, você irá escrever, testar e aprimorar programas Java. Seus programas serão mantidos em *arquivos*. Um arquivo é uma coleção de informações reunidas, por exemplo, o texto de um documento redigido com um processador de texto ou as instruções de um programa Java. Arquivos têm nomes, e as regras para nomes válidos diferem entre os sistemas. Alguns sistemas permitem espaços nos nomes dos arquivos; outros não. Alguns fazem distinção entre letras maiúsculas e minúsculas; outros não. A maioria dos compiladores Java exige que arquivos Java terminem em uma *extensão* –`.java`; por exemplo, `Test.java`. Nomes de arquivos Java não podem conter espaços e a distinção entre letras maiúsculas e minúsculas é importante.

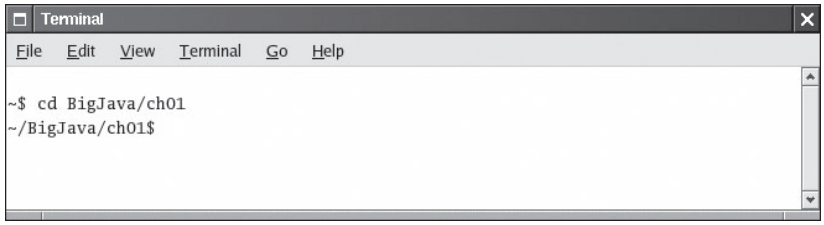


Figura 7 Uma janela de shell.

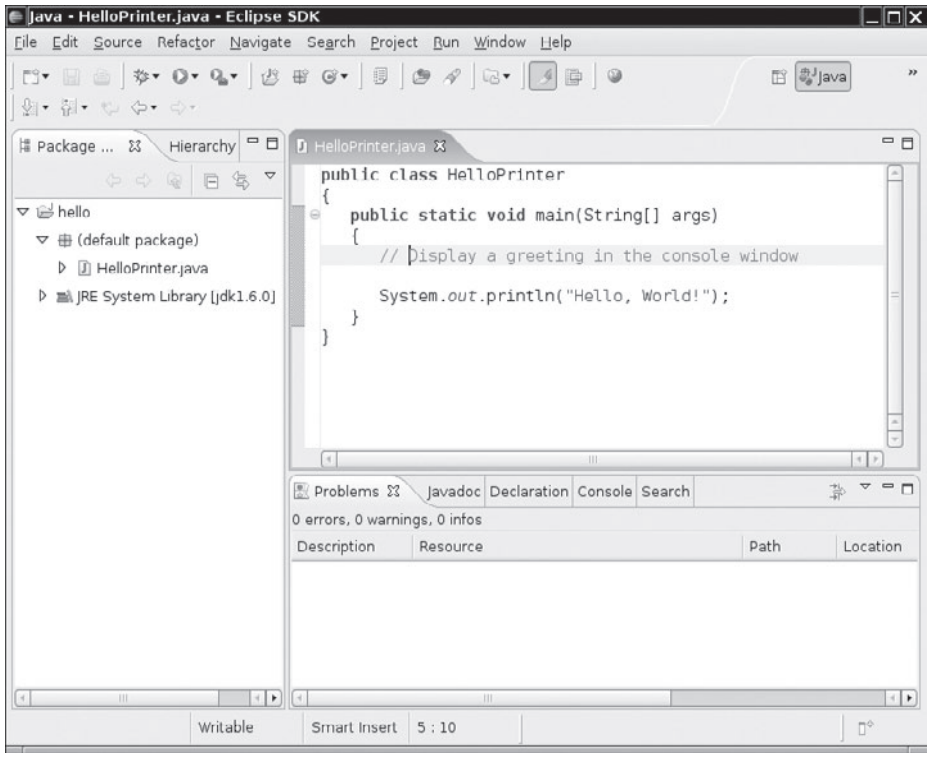


Figura 8 Um ambiente de desenvolvimento integrado.

Os arquivos são armazenados em *pastas* ou *diretórios*. Esses contêineres de arquivos podem estar *aninhados*. Isto é, uma pasta pode conter não apenas arquivos como também outras pastas, que podem conter mais arquivos e pastas (veja Figura 9). Essa hierarquia

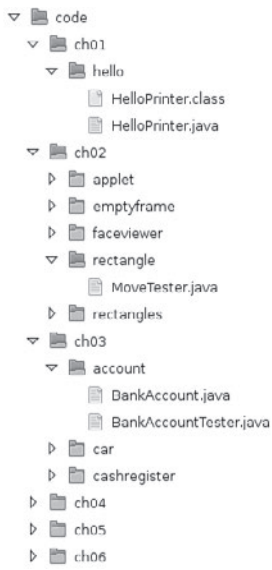


Figura 9
Pastas aninhadas.

pode ser bem ampla, especialmente nos computadores em rede, nos quais alguns arquivos podem estar no seu disco local e outros em algum outro lugar na rede. Embora não precise se preocupar com cada ramificação da hierarquia, você deve se familiarizar com o seu ambiente local. Diferentes sistemas têm diferentes maneiras de exibir arquivos e diretórios. Alguns usam uma exibição gráfica e permitem que você se movimente clicando com o mouse nos ícones de pasta. Em outros sistemas, você precisa inserir comandos para acessar ou inspecionar locais diferentes.

Passo 4. Escreva um programa simples

Na próxima seção, introduziremos um programa bem simples. Você precisará aprender a digitá-lo, executá-lo e corrigir enganos.

Passo 5. Salve seu trabalho

Desenvolva uma estratégia para manter cópias de backup do seu trabalho antes que desastres aconteçam.

Você passará várias horas digitando e melhorando o código de programas Java. Os arquivos de programa resultantes têm valor e você deve tratá-los como trataria outros bens importantes. Uma estratégia conscienciosa de segurança é especialmente importante para os arquivos de computador. Eles são mais frágeis do que documentos em papel ou outros objetos mais tangíveis. É fácil excluir um arquivo acidentalmente e às vezes perdemos arquivos por causa de um mau funcionamento do computador. Só se houver uma cópia, você não precisará redigitar o conteúdo. Como talvez você não consiga lembrar de todo o arquivo, provavelmente gastará quase tanto tempo quanto gastou para digitá-lo e aprimorá-lo. Isso custa tempo e pode fazer com que você perca prazos finais de entrega. Portanto, é crucial aprender a salvaguardar os arquivos e a ter o hábito de fazer isso *antes* de o desastre acontecer. Você pode fazer cópias de segurança, ou *backups*, dos arquivos salvando cópias em um disquete ou CD, em uma outra pasta, na sua rede local ou na Internet.

Autoverificação da aprendizagem

10. Como os projetos de programação são armazenados em um computador?
11. O que fazer para se proteger contra a perda de dados quando você trabalha nos projetos de programação?

Dica de produtividade 1.1

Entenda o sistema de arquivos

Nos últimos anos, o uso de computadores ficou mais fácil para usuários domésticos ou administrativos. Muitos detalhes não-essenciais estão agora escondidos dos usuários casuais. Por exemplo, vários usuários casuais simplesmente armazenam todo o trabalho dentro de uma pasta padrão (como “Home” ou “Meus Documentos”) e ignoram os detalhes do sistema de arquivos.

Mas você precisa saber impor uma organização para os dados que cria. Você também precisa ser capaz de localizar e inspecionar os arquivos necessários para traduzir e executar programas Java.

Se você não se sentir confortável com arquivos e pastas, certifique-se de investir algum tempo para aprender esses conceitos. Matricule-se em um curso de curta duração ou faça um tutorial na Web. Há muitos tutoriais gratuitos disponíveis na Internet, mas infelizmente suas localizações mudam com freqüência. Pesquise na Web "tutoriais para arquivos e pastas" e selecione um que vá além dos princípios básicos.

DICA DE PRODUTIVIDADE 1.2

Tenha uma estratégia de backup

A Dica de Produtividade 1.2 discute estratégias para fazer o backup do seu trabalho de programação a fim de assegurar que os dados não sejam comprometidos caso o computador sofra alguma avaria.

1.6 Compilando um programa simples

Agora você está pronto para escrever e executar seu primeiro programa Java. A escolha tradicional do primeiro programa em uma nova linguagem de programação é a de um que exibe uma saudação simples: "Hello, World!". Vamos seguir essa tradição. Eis o programa "Hello, World!" em Java.

ch01/hello/HelloPrinter.java

```
1  public class HelloPrinter
2  {
3     public static void main(String[] args)
4     {
5        // Exibe uma saudação na janela da console
6
7        System.out.println("Hello, World!");
8     }
9  }
```

Saída

```
Hello, World!
```

Java diferencia maiúsculas de minúsculas. Você precisa tomar cuidado e distingui-las corretamente.

Examinaremos esse programa logo a seguir. Por enquanto, você dever criar um novo arquivo de programa e chamá-lo `HelloPrinter.java`. Insira as instruções do programa, compile e execute-o, seguindo o procedimento apropriado para seu compilador.

Java diferencia letras maiúsculas de minúsculas. Você deve inserir letras maiúsculas e minúsculas exatamente como elas aparecem na listagem do programa. Você não pode digitar `MAIN` ou `PrintLn`. Se não for cuidadoso, terá problemas – veja Erro Comum 1.2.

Por outro lado, a organização de Java tem *formato livre*. Você pode utilizar espaços e quebras de linha para separar as palavras e pode colocar tantas palavras quanto possível em uma linha:

```
public class HelloPrinter{public static void main(String[]
args){// Exibe uma saudação na janela de console
System.out.println("Hello, World!");}}
```

Você também pode escrever cada palavra e cada símbolo em uma linha separada:

```
public
class
HelloPrinter
{
public
static
void
main
(
. . .
```

Organize seus programas de modo que possam ser lidos facilmente.

Mas um bom estilo determina que você deve organizar seus programas de uma maneira legível. Forneceremos recomendações para uma boa organização ao longo de todo o livro. O Apêndice A contém um resumo das nossas recomendações.

Quando você executar o programa de teste, a mensagem:

```
Hello, World!
```

aparecerá em algum lugar na tela (veja Figuras 10 e 11). A localização exata depende do seu ambiente de programação.

Agora que vimos o programa em funcionamento, é hora de entender sua constituição. A primeira linha:

```
public class HelloPrinter
```

Classes são os blocos de construção fundamentais dos programas Java.

inicia uma nova *classe*. Classes são um conceito fundamental em Java e você começará a estudá-las no Capítulo 2. Em Java, cada programa consiste em uma ou mais classes.

A palavra-chave `public` indica que a classe pode ser utilizada pelo "público". Mais tarde você encontrará recursos `private`. Nesse ponto, você simplesmente deve considerar:

```
public class NomeDaClasse
{
   . . .
}
```

como uma parte necessária da estrutura exigida para escrever qualquer programa Java. Em Java, cada arquivo-fonte pode conter no máximo uma classe pública, e o nome dessa classe pública deve corresponder ao nome do arquivo que a contém. Por exemplo, a classe `HelloPrinter` *precisa* estar contida em um arquivo `HelloPrinter.java`. É muito importante que os nomes e o *uso de letras maiúsculas e minúsculas* correspondam exatamente. Você pode receber mensagens de erro estranhas se atribuir à classe um nome `HELLOPrinter` ou `helloprinter.java` ao arquivo.

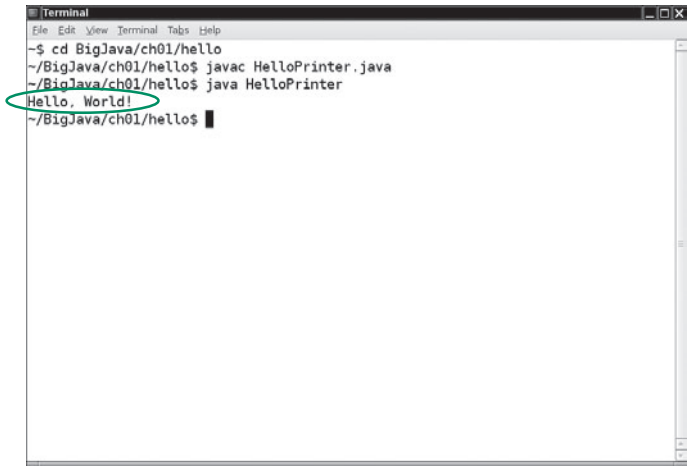


Figura 10 Executando o programa `HelloPrinter` em uma janela da console.

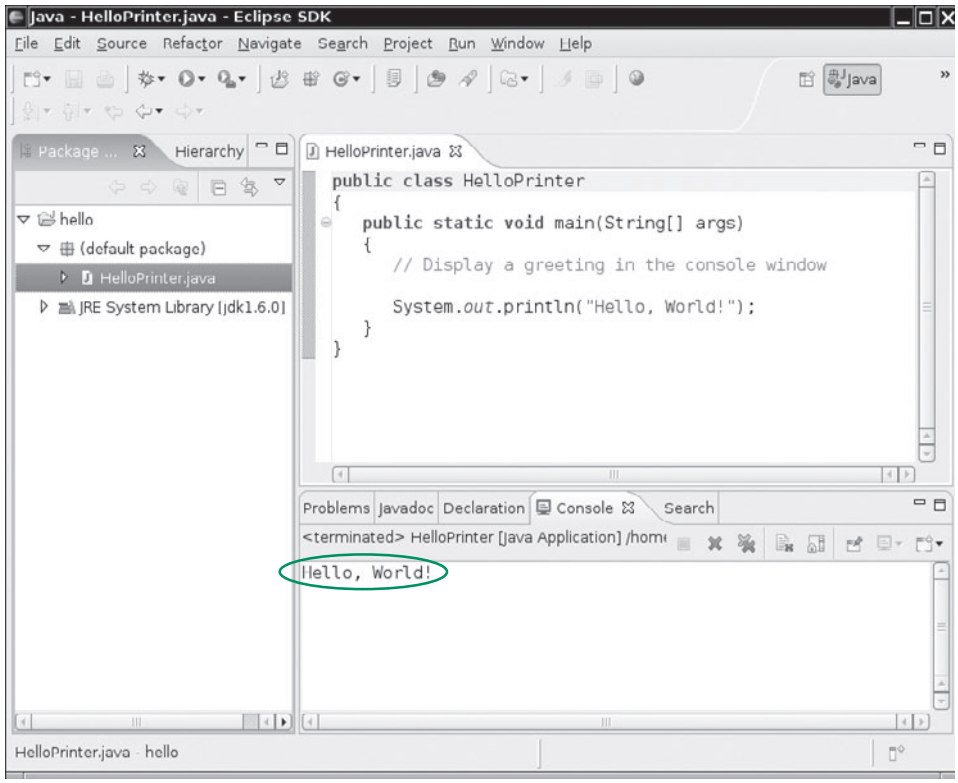


Figura 11
Executando o programa `HelloPrinter` em um ambiente de desenvolvimento integrado.

Toda aplicação Java contém uma classe com um método `main`. Quando a aplicação inicia, as instruções no método `main` são executadas.

Cada classe contém definições de métodos. Cada método contém uma seqüência de instruções.

A construção

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

define um *método* chamado `main`. Um método contém um conjunto de instruções de programação que descreve como executar uma tarefa particular. Toda aplicação Java deve ter um método `main`. A maioria dos programas Java contém outros métodos além do `main`, e você verá no Capítulo 3 como escrever outros métodos.

O *parâmetro* `String[] args` é uma parte obrigatória do método `main`. (Ele contém os *argumentos da linha de comando*, que só discutiremos no Capítulo 11.) A palavra-chave `static` indica que o método `main` não opera sobre um *objeto*. (Como veremos no Capítulo 2, a maioria dos métodos em Java opera sobre objetos, e métodos `static` não são comuns em grandes programas Java. Contudo, `main` sempre dever ser `static`, porque ele começa a executar antes de o programa criar objetos.)

Nesse momento, simplesmente considere

```
public class NomeDaClasse
{
   public static void main(String[] args)
   {
      . . .
   }
}
```

como outra parte do código básico necessário. Nosso primeiro programa tem todas as suas instruções dentro do método `main` de uma classe.

A primeira linha dentro do método `main` é um *comentário*

```
// Exibe uma saudação na janela da console
```

Use comentários para ajudar as pessoas a entender seu programa.

Esse comentário é simplesmente para o benefício de quem está lendo o programa e destina-se a explicar em mais detalhes o que a próxima instrução faz. Qualquer texto incluído entre `//` e o fim da linha é completamente ignorado pelo compilador. Os comentários são utilizados para explicar o programa a outros programadores ou para você mesmo.

As instruções ou os *comandos* no *corpo* do método `main` – isto é, as instruções dentro das chaves (`{}`) – são executadas uma a uma. Cada instrução termina em um ponto-e-vírgula (`;`). Nosso método tem uma única instrução:

```
System.out.println("Hello, World!");
```

Essa instrução imprime uma linha de texto, a saber, "Hello, World!". Mas há vários locais para os quais um programa pode enviar essa string: uma janela, um arquivo ou um computador em rede no outro lado do mundo. Você precisa especificar que o destino para a string é a *saída padrão do sistema* – isto é, uma janela de console. A janela de console é representada em Java por um objeto chamado `out`. Assim como você precisou colocar o método `main` em uma classe `HelloPrinter`, os projetistas da biblioteca Java precisaram colocar o objeto `out` em uma classe. Eles o colocaram na classe `System`, que contém objetos e métodos úteis para acessar recursos do sistema. Para utilizar o objeto `out` na classe `System`, você deve referenciá-lo como `System.out`.

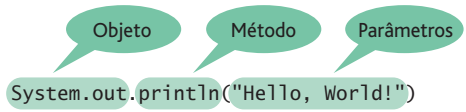


Figura 12 Chamando um método.

Para utilizar um objeto, como `System.out`, especifique o que você quer fazer com ele. Nesse caso, você quer imprimir uma linha de texto. O método `println` executa essa tarefa.

Você não é obrigado a implementar esse método – os programadores que escreveram a biblioteca Java já fizeram isso para nós – mas você precisa *chamar* o método.

Chama-se um método especificando um objeto, o nome do método e os seus parâmetros.

Sempre que você chama um método em Java, você precisa especificar três itens (veja Figura 12):

1. O objeto que você quer utilizar (nesse caso, `System.out`).
2. O nome do método que você quer utilizar (nesse caso, `println`).
3. Um par de parênteses, contendo qualquer outra informação que o método precise (nesse caso, `"Hello, World!"`). O termo técnico para essa informação é um *parâmetro* para o método. Observe que os dois pontos em `System.out.println` têm significados diferentes. O primeiro ponto significa "localize o objeto `out` na classe `System`". O segundo ponto significa "aplique o método `println` a esse objeto".

Uma seqüência de caracteres incluída entre aspas:

```
"Hello, World!"
```

Uma string é uma seqüência de caracteres incluídos entre aspas.

é chamada de *string*. Você deve incluir o conteúdo da string dentro de aspas para que o compilador saiba que você literalmente quer dizer `"Hello, World!"`. Há uma razão para essa exigência. Suponha que você precise imprimir a palavra *main*. Incluindo-a entre aspas, `"main"`, o compilador sabe que você tem em mente a seqüência de caracteres `m a i n`, não o método chamado `main`. A regra é simplesmente incluir todas as strings de texto entre aspas para que o compilador considere-as como texto simples e não tente interpretá-las como instruções de programa.

Você também pode imprimir valores numéricos. Por exemplo, a instrução

```
System.out.println(3 + 4);
```

exibe o número 7.

O método `println` imprime uma string ou um número e então inicia uma nova linha. Por exemplo, a seqüência de instruções

```
System.out.println("Hello");
System.out.println("World!");
```

imprime duas linhas de texto:

```
Hello
World!
```

Há um segundo método, chamado `print`, que você pode utilizar para imprimir um item sem iniciar uma nova linha. Por exemplo, a saída das duas instruções

```
System.out.print("00");
System.out.println(3 + 4);
```

é a única linha

```
007
```

SINTAXE 1.1 Chamada de método

objeto.nomeDoMétodo(parâmetros)

Exemplo:

```
System.out.println("Hello, Dave!")
```

Objetivo:

Invocar um método em um objeto e fornecer qualquer parâmetro adicional

AUTOVERIFICAÇÃO DA APRENDIZAGEM

12. Como você modificaria o programa `HelloPrinter` para imprimir as palavras "Hello" e "World!" em duas linhas?
13. O programa continuaria a funcionar se você omitisse a linha que inicia com `//`?
14. O que o conjunto de instruções a seguir imprime?

```
System.out.print("My lucky number is");
System.out.println(3 + 4 + 5);
```

ERRO COMUM 1.1

Omitindo ponto-e-vírgulas

Em Java, cada instrução deve terminar em um ponto-e-vírgula. Esquecer de digitar um ponto-e-vírgula é um erro comum. Isso confunde o compilador, porque ele usa o ponto-e-vírgula para descobrir onde uma instrução termina e a próxima inicia. O compilador não usa quebras de linha nem fechamento de chaves para reconhecer o fim de uma instrução. Por exemplo, o compilador considera

```
System.out.println("Hello")
System.out.println("World!");
```

uma única instrução, como se você tivesse escrito:

```
System.out.println("Hello") System.out.println("World!");
```

Assim, o compilador não entende essa instrução, porque ele não esperava a palavra `System` após o parêntese de fechamento depois de `"Hello"`. A solução é simples. Verifique todas as instruções para ver se terminam com ponto-e-vírgula, da mesma maneira como você verificaria um ponto final em uma frase qualquer.

TÓPICO AVANÇADO 1.1

Sintaxe alternativa de comentário

Em Java, há duas formas de escrever comentários. Você já aprendeu que o compilador ignora qualquer coisa que você digita entre `//` e o fim da linha atual. O compilador também ignora qualquer texto entre um `/*` e `*/`.

```
/* Um programa Java simples */
```

O comentário `//` é mais fácil de digitar se ele tiver um comprimento de apenas uma única linha. Se um comentário tiver mais que uma linha, utilize a notação `/* . . . */`:

```
/*
    Este é um programa Java simples que você pode utilizar para testar o
    compilador e a máquina virtual.
*/
```

Seria entediante adicionar o `//` ao começo de cada linha e movê-lo sempre que o texto do comentário mudar.

Neste livro, utilizamos `//` para comentários que nunca ultrapassarão uma linha e `/* . . . */` para comentários mais longos. Se preferir, você pode utilizar sempre o estilo `//`. Os leitores do seu código ficarão gratos por *qualquer* comentário, independentemente do estilo que você usa.

1.7 Erros

Vamos fazer algumas experiências com o programa `HelloPrinter`. O que acontece se você cometer um erro de digitação como:

```
System.ouch.println("Hello, World!");
System.out.println("Hello, World!);
System.out.println("Hello, Word!");
```

Um erro de sintaxe é uma violação das regras da linguagem de programação. O compilador detecta erros de sintaxe.

No primeiro caso, o compilador reclamará. Ele informará que não sabe o que você quer dizer com `ouch`. O texto exato da mensagem de erro depende do compilador, mas poderia ser algo como "Undefined symbol ouch". Esse é um *erro em tempo de compilação* ou *erro de sintaxe*. De acordo com as regras da linguagem, algo está errado e o compilador localiza esse erro. Quando o compilador encontra um ou mais erros, ele se recusa a traduzir o programa em instruções para a máquina virtual Java e, conseqüentemente, não haverá programa para você executar. Você deve corrigir o erro e compilar novamente. Na realidade, o compilador é bem exigente e é comum passar por várias rodadas de correção de erros em tempo de compilação antes de a compilação ser bem-sucedida pela primeira vez.

Se o compilador localizar um erro, ele não irá simplesmente parar e desistir. Ele tentará informar o maior número de erros que puder localizar para que você possa corrigir todos de uma só vez. Às vezes, porém, o compilador identifica um erro não onde ele realmente está. É isso o que acontece com o erro na segunda linha. Como a aspa de fechamento não está presente, o compilador achará que os caracteres `);` ainda são parte da string. Nesses casos, é comum o compilador emitir informes de erro falsos em linhas

próximas. Você só deve corrigir as mensagens de erro que fazem sentido para você e então recompilar.

O erro na terceira linha é de um tipo diferente. O programa irá compilar e executar, mas sua saída estará errada. Ele imprimirá:

```
Hello, Word!
```

Um erro de lógica faz o programa agir diferentemente do que pretendia o programador. Você deve testar seus programas para encontrar erros de lógica.

Esse é um *erro em tempo de execução* ou *erro de lógica*. O programa está sintaticamente correto e realiza algo, mas ele não faz o que supostamente deveria fazer. O compilador não pode localizar o erro. Você, o programador, deve remover esse tipo de erro. Execute o programa e examine a saída cuidadosamente.

Durante o desenvolvimento de um programa, erros são inevitáveis. Quando um programa tem mais do que algumas linhas de comprimento, exige-se muita concentração para digitá-lo corretamente, sem cometer erros. Omitir ponto-e-vírgulas ou aspas é mais comum do que se imagina, mas o compilador irá rastrear esses problemas para você.

Erros de lógica são mais problemáticos. O compilador não os encontrará – de fato, o compilador traduzirá sem problemas qualquer programa contanto que a sintaxe esteja correta – mas o programa resultante fará algo errado. É responsabilidade do autor do programa testá-lo e localizar todos os erros de lógica. O teste de programas é um tópico importante que veremos muitas vezes neste livro. Outro aspecto importante da boa prática é *a programação defensiva:* estruturar programas e processos de desenvolvimento de forma que um erro em uma das partes de um programa não desencadeie uma resposta desastrosa.

Os exemplos de erros que vimos até agora não foram difíceis de diagnosticar ou corrigir, mas, à medida que você aprende técnicas de programação mais sofisticadas, também haverá muito mais espaço para erros. O fato desagradável é que localizar todos os erros em um programa é muito difícil. Mesmo se você puder observar que um programa exibe um comportamento falho, talvez não seja tão óbvio descobrir qual parte do programa causou esse comportamento e como você pode corrigi-lo. Ferramentas especiais de software (chamadas *depuradores*) permitem investigar um programa para localizar *bugs* – isto é, erros de lógica. No Capítulo 6 você aprenderá a utilizar um depurador eficientemente.

Observe que esses erros são diferentes do tipo de erro que talvez você cometa em cálculos manuais. Ao somar uma coluna de números, por exemplo, você poderia deixar de acrescentar um sinal de subtração ou esquecer o "vai um", talvez porque esteja entediado ou cansado. Computadores não cometem esse tipo de erro.

Este livro usa uma estratégia de gerenciamento de erro dividida em três partes. Primeiro, você aprenderá os erros comuns e como evitá-los. Depois aprenderá as estratégias de programação defensiva para minimizar a probabilidade e o impacto dos erros. Por fim, você aprenderá a estratégia de depuração para remover os erros que restarem.

Autoverificação da aprendizagem

15. Suponha que você omita os caracteres `//` do programa `HelloPrinter.java`, mas não o restante do comentário. Você receberá um erro em tempo de compilação ou em tempo de execução?
16. Como você pode encontrar erros de lógica em um programa?

ERRO COMUM 1.2

Palavras digitadas incorretamente

Se você digitar uma palavra errada acidentalmente, coisas estranhas podem acontecer e nem sempre é completamente óbvio, a partir das mensagens de erro, saber o que deu errado. Eis um bom exemplo de como erros simples de ortografia podem causar problemas:

```
public class HelloPrinter
{
   public static void Main(String[] args)
   {
      System.out.println("Hello, World!");
   }
}
```

Essa classe define um método chamado `Main`. O compilador não irá considerar isso como sendo o mesmo que o método `main`, porque `Main` inicia com letra maiúscula e a linguagem Java diferencia letras maiúsculas de minúsculas. O compilador diferencia maiúsculas de minúsculas, portanto, `Main` é tão diferente de `main` quanto `rain`. O compilador irá compilar seu método `Main`, mas ao ler o arquivo compilado, a máquina virtual Java irá reclamar a ausência do método `main` e não executará o programa. Naturalmente, a mensagem "método main ausente" deve fornecer uma pista sobre onde procurar o erro.

Se receber uma mensagem de erro que aparentemente indica que o compilador está na trilha errada, uma boa idéia é verificar a ortografia e o uso de letras maiúsculas e minúsculas. Todas as palavras-chave do Java utilizam apenas letras minúsculas. Nomes de classe normalmente iniciam com letra maiúscula, nomes de método e variáveis com letra minúscula. Se digitar incorretamente o nome de um símbolo (por exemplo, `ouch` em vez de `out`), o compilador reclamará quanto a um "símbolo indefinido". Essa mensagem de erro normalmente é um bom indício de que você cometeu um erro de ortografia.

1.8 O processo de compilação

Alguns ambientes de desenvolvimento Java são muito convenientes de usar. Insira o código em uma janela, clique em um botão para compilar e clique em outro botão para executar o programa. As mensagens de erro aparecem em uma segunda janela e o programa executa em uma terceira janela. Com esse tipo de ambiente você fica completamente protegido dos detalhes do processo de compilação. Em outros sistemas você deve executar cada passo manualmente, digitando comandos em uma janela de shell.

Um editor é um programa que permite inserir e modificar texto, por exemplo, um programa Java.

Independentemente do ambiente de compilação utilizado, você começa sua atividade digitando as instruções do programa. O software que você usa para inserir e modificar o texto do programa é chamado *editor*. Lembre-se de *salvar* seu trabalho em disco freqüentemente, pois, do contrário, o editor de textos só armazenará o texto na memória do computador. Se algo der errado com o computador e você precisar reiniciá-lo, o conteúdo da memória principal (incluindo o texto do programa digitado)

será perdido, mas qualquer coisa armazenada no disco rígido é permanente, mesmo se você precisar reiniciar o computador.

O compilador Java traduz código-fonte em arquivos de classe que contêm instruções para a máquina virtual Java.

Quando você compila seu programa, o compilador traduz o *código-fonte* Java (isto é, as instruções que você escreveu) em *arquivos de classe*, que consistem em instruções para a máquina virtual e outras informações necessárias para a execução. Os arquivos de classe têm a extensão `.class`. Por exemplo, as instruções para a máquina virtual referentes ao programa `HelloPrinter` são armazenadas no arquivo `HelloPrinter.class`. Como já mencionado, o compilador só produz um arquivo de classe depois que você corrigiu todos os erros de sintaxe.

O arquivo de classe contém apenas a tradução das instruções que você escreveu. Isso não é suficiente para executar o programa. Para exibir uma string em uma janela é necessário um bom número de atividades de baixo nível. Os autores das classes `System` e `PrintStream` (que define o objeto `out` e o método `println`) implementaram todas as ações necessárias e armazenaram os arquivos de classe necessários em uma *biblioteca*. Uma biblioteca é uma coleção de código que foi programada e traduzida por uma outra pessoa, e está pronta para ser utilizada em seu programa.

A máquina virtual Java carrega as instruções para o programa que você escreveu, inicia seu programa e carrega os arquivos de biblioteca necessários à medida que são exigidos.

Os passos da compilação e execução do seu programa estão esboçados na Figura 13.

A máquina virtual Java carrega instruções de programa a partir dos arquivos de classe e dos arquivos de biblioteca.

A atividade de programação segue a seguinte seqüência de passos. Iniciar o editor e escrever o arquivo-fonte. Compilar o programa e examinar as mensagens de erro. Voltar ao editor e corrigir os erros de sintaxe. Quando o compilador concluir com sucesso, executar o programa. Se encontrar um erro em tempo de execução, você deverá examinar o código-fonte no editor para tentar determinar a razão desse erro.

Depois de encontrar a causa do erro, corrija-o no editor. Compile e execute novamente para ver se o erro desapareceu. Do contrário, volte ao editor. Isso é chamado ciclo *editar–compilar–testar* (veja Figura 14). Você passará muito tempo nesse ciclo ao trabalhar com os exercícios de programação.

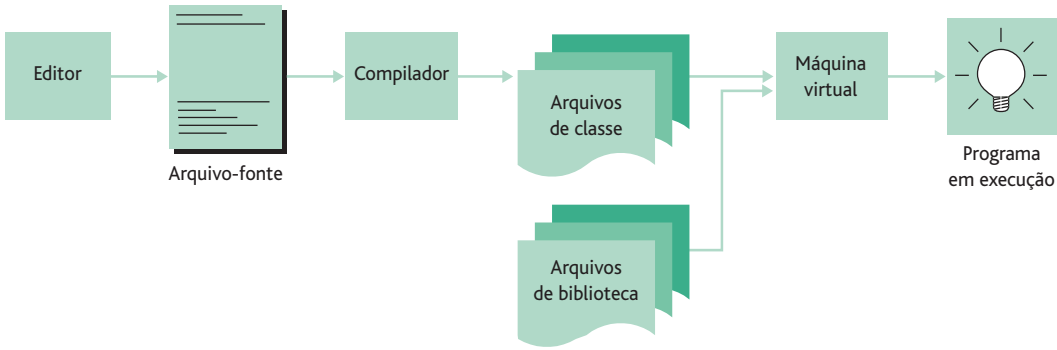


Figura 13 Do código-fonte à execução do programa.

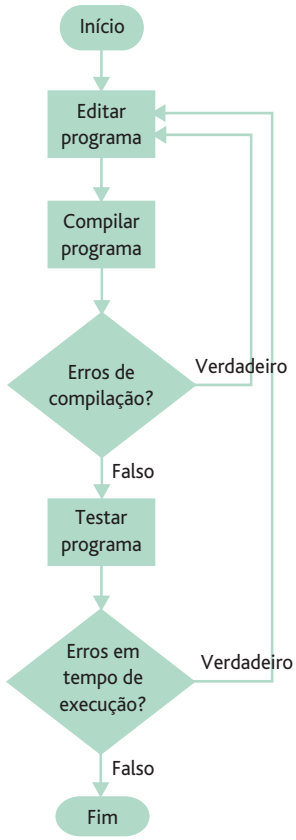


Figura 14 O ciclo editar-compilar-testar.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

17. O que você espera ver quando carrega um arquivo de classe no seu editor de texto?
18. Por que você não pode testar erros em tempo de execução em um programa quando ele tem erros de compilação?

RESUMO DO CAPÍTULO

1. Um computador precisa ser programado para que realize tarefas. Diferentes tarefas requerem diferentes programas.
2. Um programa de computador executa uma seqüência de operações muito básicas em rápida sucessão.
3. Um programa de computador contém as seqüências de instruções para todas as tarefas que pode executar.
4. No coração do computador reside a Unidade Central de Processamento (*Central Processing Unit* – CPU).
5. Dados e programas são mantidos na unidade de armazenamento primário (memória) e nas unidades de armazenamento secundário (por exemplo, um disco rígido).

6. A CPU lê as instruções de máquina a partir da memória. As instruções a orientam a se comunicar com a memória, com as unidades de armazenamento secundário e com os dispositivos periféricos.
7. Geralmente, o código de máquina depende do tipo de CPU. Entretanto, o conjunto de instruções da máquina virtual Java (Java Virtual Machine – JVM) pode ser executado em diferentes CPUs.
8. Uma vez que as instruções de máquina são codificadas como números, é difícil escrever programas em código de máquina.
9. Linguagens de alto nível permitem descrever tarefas em um nível conceitual mais alto que o do código de máquina.
10. Um compilador traduz programas escritos em uma linguagem de alto nível em código de máquina.
11. Java foi originalmente projetado para programar dispositivos de consumo popular, mas foi inicialmente utilizado com sucesso para escrever applets para Internet.
12. Java foi projetado para ser seguro e portável, beneficiando tanto usuários da Internet como estudantes.
13. Java tem uma biblioteca bastante extensa. Concentre o aprendizado naquelas partes da biblioteca que você precisa para seus projetos de programação.
14. Reserve algum tempo para se familiarizar com o sistema computacional e o compilador Java que utilizará para seus trabalhos escolares.
15. Desenvolva uma estratégia para manter cópias de backup do seu trabalho antes que desastres aconteçam.
16. Java diferencia letras maiúsculas de minúsculas. Tenha cuidado para também fazer essa distinção.
17. Organize seus programas de modo que possam ser lidos facilmente.
18. As classes são os blocos de construção fundamentais dos programas Java.
19. Cada aplicação Java contém uma classe com um método `main`. Quando a aplicação inicia, as instruções no método `main` são executadas.
20. Cada classe contém definições de métodos. Cada método contém uma seqüência de instruções.
21. Utilize comentários para que outras pessoas possam entender seu programa.
22. Um método é chamado especificando-se um objeto, o nome do método e os parâmetros do método.
23. Uma string é uma seqüência de caracteres entre aspas.
24. Um erro de sintaxe é uma violação das regras da linguagem de programação. O compilador detecta erros de sintaxe.
25. Um erro de lógica faz um programa tomar uma ação que o programador não pretendia. Você deve testar seus programas para encontrar erros de lógica.
26. Um editor é um programa para inserir e modificar texto, como por exemplo, um programa Java.
27. O compilador Java traduz código-fonte em arquivos de classe que contêm instruções para a máquina virtual Java.
28. A máquina virtual Java carrega instruções de programa a partir de arquivos de classe e de arquivos de biblioteca.

LEITURA ADICIONAL

1. `http://jmol.sourceforge.net/applet/` Site da applet jmol para visualizar moléculas.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

Eis uma lista de todos os métodos, classes, variáveis estáticas e constantes introduzidos neste capítulo. Consulte a documentação no Apêndice C para informações adicionais.

```
java.io.PrintStream
   print
   println
java.lang.System
   out
```

EXERCÍCIOS DE REVISÃO

★ **Exercício R1.1.** Explique a diferença entre usar um programa de computador e programar um computador.

★ **Exercício R1.2.** O que distingue um computador de um eletrodoméstico comum?

★ **Exercício R1.3.** Classifique os dispositivos de armazenamento que podem ser parte de um sistema computacional de acordo com:

a. Velocidade
b. Preço
c. Capacidade de armazenamento

★★ **Exercício R1.4.** O que é a máquina virtual Java?

★ **Exercício R1.5.** O que é uma applet?

★ **Exercício R1.6.** O que é um ambiente integrado de programação?

★ **Exercício R1.7.** O que é uma janela de console?

★★ **Exercício R1.8.** Descreva *exatamente* quais passos você seguiria para fazer o backup do seu trabalho depois de digitar o programa `HelloPrinter.java`.

★★ **Exercício R1.9.** No seu próprio computador ou no computador de um laboratório, encontre a localização exata (nome de pasta ou nome de diretório) do:

a. Arquivo de exemplo `HelloPrinter.java`, que você escreveu com o editor
b. Dispositivo para carregamento do programa Java, `java.exe` ou `java`
c. Arquivo de biblioteca `rt.jar` que contém a biblioteca de tempo de execução

★ **Exercício R1.10.** Como você descobre erros de sintaxe? Como você descobre erros de lógica?

★★ **Exercício R1.11.** Escreva três versões do programa `HelloPrinter.java` que contenham diferentes erros de sintaxe. Escreva uma versão que contenha um erro de lógica.

★★★ **Exercício R1.12.** O que as instruções a seguir imprimem? Não tente adivinhar; escreva programas para descobrir.

a. `System.out.println("3 + 4");`

b. `System.out.println(3 + 4);`

c. `System.out.println(3 + "4");`

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P1.1.** Escreva um programa `NamePrinter` que exibe seu nome dentro de uma caixa na tela de console, como mostrado aqui:

```
+-----+
|Dave|
+-----+
```

Faça o melhor possível para aproximar linhas com caracteres, como `|`, `-` e `+`.

★ **Exercício P1.2.** Escreva um programa `FacePrinter` que imprima um rosto, utilizando caracteres de texto, com uma aparência melhor do que esta:

```
  /////
 |  o o  |
(|   ^   |)
 |  [_]  |
  -----
```

Utilize *comentários* para indicar as instruções que imprimem o cabelo, as orelhas, a boca e assim por diante.

★ **Exercício P1.3.** Escreva um programa `TicTacToeBoardPrinter` que imprime um quadro de jogo da velha:

```
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
|   |   |   |
+---+---+---+
```

★ **Exercício P1.4.** Escreva um programa `StaircasePrinter` que imprime uma escada:

```
            +---+
            |   |
        +---+---+
        |   |   |
    +---+---+---+
    |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
```

★★ **Exercício P1.5.** Escreva um programa que calcule a soma dos primeiros dez inteiros positivos, $1 + 2 + \cdots + 10$. *Dica*: Escreva um programa seguindo este formato:

```
public class Sum10
{
   public static void main(String[] args)
   {
      System.out.println(          );
   }
}
```

★★ **Exercício P1.6.** Escreva um programa `Sum10Reciprocals` que calcule a soma dos recíprocos $1/1 + 1/2 + \cdots + 1/10$. Isso é mais difícil do que parece. Tente escrever o programa e verifique o resultado. O resultado do programa provavelmente estará incorreto. Escreva então os denominadores como *números de ponto flutuante*, `1.0`, `2.0`,..., `10.0` e execute o programa novamente. Você consegue explicar a diferença nos resultados? Iremos explorar esse fenômeno no Capítulo 4.

★★ **Exercício P1.7.** Digite e execute o programa a seguir:

```
import javax.swing.JOptionPane;

public class DialogViewer
{
   public static void main(String[] args)
   {
      JOptionPane.showMessageDialog(null, "Hello, World!");
      System.exit(0);
   }
}
```

Então modifique o programa para que mostre a mensagem "Hello, *seu nome*!"

★★ **Exercício P1.8.** Digite e execute o programa a seguir:

```
import javax.swing.JOptionPane;

public class DialogViewer
{
   public static void main(String[] args)
   {
      String name = JOptionPane.showInputDialog("What is your name?");
      System.out.println(name);
      System.exit(0);
   }
}
```

Modifique mais uma vez o programa para que ele imprima "Hello *nome*!", exibindo o nome que o usuário digitou.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 1.1.** Este projeto aprimora os Exercícios P1.7 e P1.8. Seu programa deve ler o nome do usuário e então mostrar uma seqüência de duas caixas de diálogo:

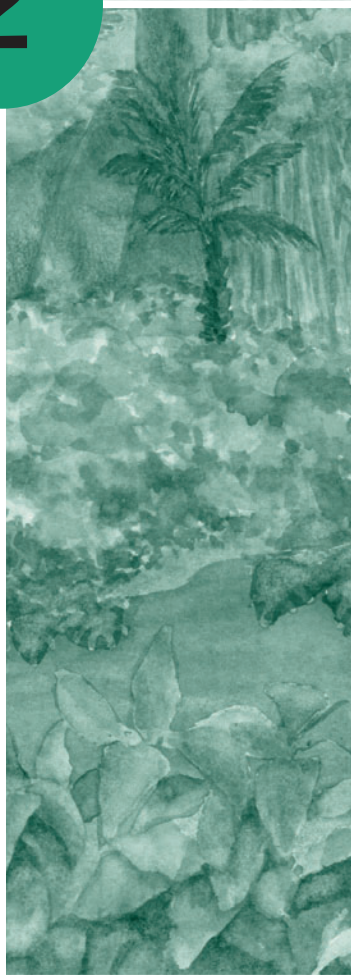
- Primeiro, uma caixa de diálogo de entrada que pergunta: "O que você quer que eu faça?"
- Então uma caixa de diálogo de mensagem que diz: "Sinto muito, (*seu nome*). Acho que não posso fazer isso."

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Um programa que lê os dados no CD e envia a saída para os alto-falantes e a tela.
2. Um CD player pode fazer uma única coisa – tocar CDs de música. Ele não pode executar programas.
3. Não – esse programa simplesmente executa as seqüências de instrução que os programadores prepararam antecipadamente.
4. No armazenamento secundário, em geral um disco rígido.
5. A unidade central de processamento.
6. 21 100
7. Não – um compilador destina-se a programadores para traduzir instruções de programa de alto nível em código de máquina.
8. Segurança e portabilidade.
9. Ninguém consegue aprender a biblioteca inteira – ela é muito grande.
10. Os programas são armazenados em arquivos e os arquivos são armazenados em pastas ou diretórios.
11. Você faz o backup dos seus arquivos e pastas.
12. `System.out.println("Hello,"); System.out.println("World!");`
13. Sim – a linha que inicia com `//` é um comentário destinado aos leitores humanos. O compilador ignora comentários.
14. A impressão é `My lucky number is12`. Seria uma boa idéia adicionar um espaço depois do `is`.
15. Um erro em tempo de compilação. O compilador não saberá o que fazer com a palavra `Display`.
16. Você precisa executar o programa e observar o comportamento.
17. Uma seqüência de caracteres aleatórios, alguns com uma aparência engraçada. Os arquivos de classe contêm instruções para a máquina virtual que são codificadas como números binários.
18. Quando um programa contém erros de compilação, nenhum arquivo de classe é criado e não há nada a executar.

Capítulo 2

Utilizando Objetos



Objetivos do capítulo

- Aprender sobre variáveis
- Entender os conceitos de classes e objetos
- Ser capaz de chamar métodos
- Aprender sobre parâmetros e valores de retorno
- T Implementar programas de teste
- Ser capaz de navegar pela documentação da API
- Perceber a diferença entre objetos e referências a objeto
- G Escrever programas que exibem formas simples

A maioria dos programas úteis não manipula apenas números e strings. Lidam também com itens de dados mais complexos e que representam mais precisamente entidades no mundo real. Exemplos desses itens de dados incluem contas bancárias, registros de funcionários e formas gráficas.

A linguagem Java é ideal para projetar e manipular esses itens de dados, ou *objetos*. Em Java, você define *classes* que descrevem o comportamento desses objetos. Neste capítulo, você aprenderá a manipular objetos que pertencem a classes predefinidas. Esse conhecimento irá prepará-lo para o próximo capítulo, no qual você aprenderá a implementar suas próprias classes.

Conteúdo do Capítulo

2.1 Tipos e variáveis 60
Sintaxe 2.1: Definição de variável 61
2.2 O operador de atribuição 62
Sintaxe 2.2: Atribuição 63
2.3 Objetos, classes e métodos 63
2.4 Parâmetros de método e valores de retorno 67
2.5 Tipos numéricos 68
2.6 Construindo objetos 69
Sintaxe 2.3: Construção de objeto 71
Erro comum 2.1: Tentando invocar um construtor como se fosse um método 71
2.7 Métodos de acesso e métodos modificadores 71
2.8T Implementando um programa de teste 72
Sintaxe 2.4: Importando uma classe a partir de um pacote 74
Tópico avançado 2.1: Testando classes em um ambiente interativo
2.9 A documentação da API 75
Dica de produtividade 2.1: Não memorize – use a ajuda on-line 77
2.10 Referências a objetos 77
Fato aleatório 2.1: Mainframes – quando os dinossauros dominavam a Terra
2.11G Aplicações gráficas e janelas de frame 80
2.12G Desenhando em um componente 82
Tópico avançado 2.2: Applets
2.13G Elipses, linhas, texto e cores 85
Fato aleatório 2.2: A evolução da Internet

2.1 Tipos e variáveis

Em Java, cada valor é de um tipo.

Em Java, cada valor é de um tipo. Por exemplo, `"Hello, World"` é do tipo `String`, o objeto `System.out` é do tipo `PrintStream` e o número 13 é do tipo `int` (uma abreviação para "integer", ou inteiro). O tipo informa o que você pode fazer com os valores. Você pode chamar `println` em qualquer objeto do tipo `PrintStream`. Pode também calcular a soma ou o produto de dois inteiros quaisquer.

É muito comum querermos armazenar os valores para utilizá-los posteriormente. Para lembrar-se de um objeto, você precisa armazená-lo em uma *variável*. Uma variável é um local de armazenamento na memória do computador que possui um *tipo*, um *nome* e um conteúdo. Por exemplo, aqui declaramos três variáveis:

```
String greeting = "Hello, World!";
PrintStream printer = System.out;
int luckyNumber = 13;
```

Usa-se variáveis para armazenar valores que se deseja utilizar em um momento posterior.

A primeira variável chama-se `greeting`. Ela pode ser utilizada para armazenar valores do tipo `String` e é configurada com o valor `"Hello, World!"`. A segunda variável armazena um valor do tipo `PrintStream` e a terceira armazena um inteiro.

Variáveis podem ser utilizadas no lugar dos objetos que elas armazenam:

```
printer.println(greeting); // O mesmo que System.out.println("Hello, World!")
printer.println(luckyNumber); // O mesmo que System.out.println(13)
```

SINTAXE 2.1 Definição de variável

nomeDoTipo nomeDaVariável = *valor*;
ou
nomeDoTipo nomeDaVariável;

Exemplo:

```
String greeting = "Hello, Dave!";
```

Objetivo:

Definir uma nova variável de um tipo particular e, opcionalmente, fornecer um valor inicial

Ao declarar suas próprias variáveis, você precisa tomar duas decisões.

- Qual tipo você deve utilizar para a variável?
- Qual nome você deve atribuir à variável?

O tipo depende do uso final. Se precisar armazenar uma string, utilize o tipo `String` para sua variável.

É um erro armazenar um valor cuja classe não corresponde ao tipo da variável. Por exemplo, o seguinte é um erro:

```
String greeting = 13; // ERRO: Tipos incompatíveis
```

Você não pode utilizar uma variável `String` para armazenar um inteiro. O compilador verifica não-correspondências de tipo para protegê-lo contra erros.

Identificadores para variáveis, métodos e classes são compostos de letras, dígitos e caracteres de sublinhado.

Ao decidir sobre um nome para uma variável, você deve fazer uma escolha que descreve o propósito da variável. Por exemplo, o nome da variável `greeting` é uma escolha melhor que o nome `g`.

Um *identificador* é o nome de uma variável, método ou classe. Java impõe as seguintes regras para identificadores:

- Identificadores podem ser compostos de letras, dígitos, caracteres de sublinhado (_) e sinal de cifrão ($). Eles, porém, não podem iniciar com um dígito. Por exemplo, `greeting1` é válido, mas `1greeting` não.
- Você não pode utilizar outros símbolos como ? ou %. Por exemplo, `hello!` não é um identificador válido.
- Não são permitidos espaços em identificadores. Portanto, `lucky number` não é válido.
- Além disso, você não pode utilizar *palavras reservadas*, como `public`, como nomes; essas palavras são reservadas exclusivamente para seus significados especiais em Java.
- Identificadores também fazem *distinção entre letras maiúsculas e minúsculas*; isto é, `greeting` e `Greeting` são *diferentes*.

Por convenção, nomes de variáveis devem iniciar com uma letra minúscula.

Essas são regras rígidas da linguagem Java. Se violar uma delas, o compilador informará um erro. Além disso, há algumas *convenções* que você deve obedecer para que seus programas possam ser lidos facilmente por outros programadores:

- Nomes de variáveis e métodos devem iniciar com letra minúscula. É válido utilizar uma letra maiúscula ocasionalmente, como `luckyNumber`. Essa combinação de letras minúsculas e maiúsculas às vezes é chamada de "notação camelo" porque as letras maiúsculas se destacam como a corcova de um camelo.
- Nomes de classes devem iniciar com letra maiúscula. Por exemplo, `Greeting` seria um nome apropriado para uma classe, mas não para uma variável.

Se violar essas convenções, o compilador não reclamará, mas você irá confundir outros programadores que lêem seu código.

Autoverificação da aprendizagem

1. Qual é o tipo dos valores `0` e `"0"`?
2. Quais dos seguintes são identificadores válidos?

```
Greeting1
g
void
101dalmatians
Hello, World
<greeting>
```

3. Defina uma variável para armazenar seu nome. Utilize a notação camelo no nome da variável.

2.2 O operador de atribuição

Utilize o operador de atribuição (=) para alterar o valor de uma variável.

Você pode alterar o valor de uma variável existente com o operador de atribuição (=). Por exemplo, considere a definição da variável a seguir:

```
int luckyNumber = 13;
```
(1)

Se quiser alterar o valor dessa variável, simplesmente atribua o novo valor:

```
luckyNumber = 12;
```
(2)

A atribuição substitui o valor original da variável (veja Figura 1).

Na linguagem de programação Java, o operador = denota uma *ação*, substituir o valor de uma variável. Esse uso difere do uso tradicional do símbolo =, como um operador de igualdade.

É um erro utilizar uma variável à qual nunca foi atribuído um valor. Por exemplo, a seqüência de instruções

```
int luckyNumber;
System.out.println(luckyNumber);    // ERRO – variável não-inicializada
```

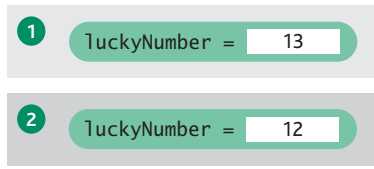
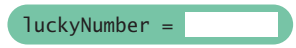


Figura 1
Atribuindo um novo valor a uma variável.

Figura 2
Uma variável objeto não-inicializada.



é um erro. O compilador reclama sobre uma "variável não-inicializada" quando você usa uma variável à qual nunca foi atribuído valor algum. (Veja Figura 2.)

Todas as variáveis devem ser inicializadas antes de você acessá-las.

O correto é atribuir um valor à variável antes de utilizá-la:

```
int luckyNumber;
luckyNumber = 13;
System.out.println(luckyNumber); // OK
```

Ou, melhor ainda, inicialize a variável ao defini-la.

```
int luckyNumber = 13;
System.out.println(luckyNumber); // OK
```

SINTAXE 2.2 Atribuição

nomeDaVariável = *valor*;

Exemplo:

```
luckyNumber = 12;
```

Objetivo:

Atribuir um novo valor a uma variável previamente definida

AUTOVERIFICAÇÃO DA APRENDIZAGEM

4. `12 = 12` é uma expressão válida na linguagem Java?
5. Como você altera o valor da variável `greeting` para `"Hello, Nina!"`?

2.3 Objetos, classes e métodos

Objetos são entidades no seu programa que você manipula invocando métodos.

Um *objeto* é uma entidade que você pode manipular no seu programa. Normalmente, você não sabe como o objeto é organizado internamente, mas ele tem um comportamento bem-definido e é isso o que nos importa quando o utilizamos.

Um método é uma seqüência de instruções que acessam os dados de um objeto.

Você manipula um objeto chamando um ou mais dos seus *métodos*. Um método consiste em uma seqüência de instruções que acessam os dados internos. Quando você chama o método, não há como saber exatamente quais são essas instruções, mas você sabe o propósito do método.

Por exemplo, vimos no Capítulo 1 que `System.out` refere-se a um objeto. Você o manipula chamando o método `println`. Quando o método `println` é chamado, algumas atividades ocorrem dentro do objeto e o efeito final é que o texto aparece na janela da console. Você não sabe como isso acontece, mas isso é válido. O importante é que o método execute aquilo que você solicitou.

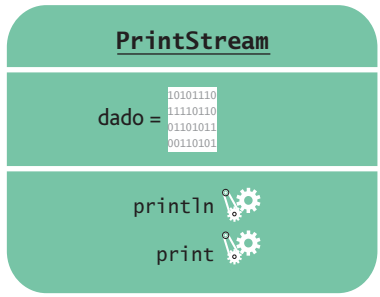


Figura 3
Representação do objeto `System.out`.

A Figura 3 mostra uma representação do objeto `System.out`. Os dados internos são simbolizados por uma seqüência de zeros e uns. Pense em cada método (simbolizado pelas engrenagens) como uma parte de maquinaria que executa uma determinada tarefa.

No Capítulo 1, você viu dois objetos:

- `System.out`
- `"Hello, World!"`

Esses objetos pertencem a diferentes *classes*. O objeto `System.out` pertence à classe `PrintStream`. O objeto `"Hello, World!"` pertence à classe `String`. Uma classe especifica os métodos que você pode aplicar aos objetos dela.

Uma classe define os métodos que você pode aplicar aos objetos dela.

Você pode utilizar o método `println` com qualquer objeto pertencente à classe `PrintStream`. `System.out` é um desses objetos. É possível obter outros objetos da classe `PrintStream`. Por exemplo, você pode construir um objeto `PrintStream` com o objetivo de enviar a saída para um arquivo. Mas só discutiremos arquivos no Capítulo 11.

Assim como a classe `PrintStream` fornece métodos como `println` e `print` para seus objetos, a classe `String` fornece métodos que você pode aplicar a objetos `String`. Um deles é o método `length`. O método `length` conta o número de caracteres em uma string. Você pode aplicar esse método a qualquer objeto do tipo `String`. Por exemplo, a seqüência de instruções

```
String greeting = "Hello, World!";
int n = greeting.length();
```

inicializa `n` com o número de caracteres do objeto `String` `"Hello, World!"`. Depois de as instruções no método `length` serem executadas, `n` é configurado como 13. (As aspas não são parte da string e o método `length` não as conta.)

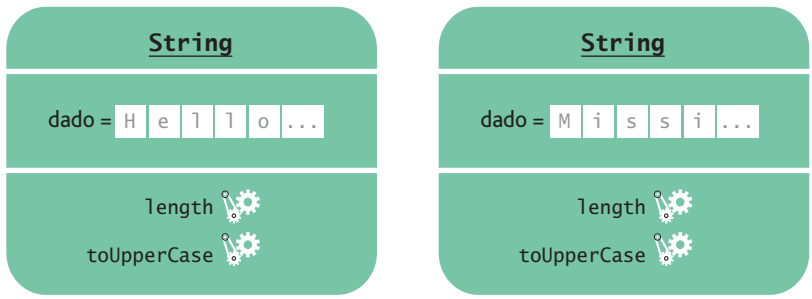


Figura 4 Uma representação de dois objetos `String`.

O método `length` – diferentemente do método `println` – não requer entrada dentro dos parênteses. Entretanto, o método `length` fornece uma saída, a saber, a contagem de caracteres.

Na próxima seção, você verá mais detalhadamente como fornecer entradas a um método e obter saídas do método.

Vejamos outro método da classe `String`. Quando você aplica o método `toUpperCase` a um objeto `String`, esse método cria outro objeto `String` que contém os caracteres da string original, com as letras minúsculas convertidas em maiúsculas. Por exemplo, a seqüência de instruções

```
String river = "Mississippi";
String bigRiver = river.toUpperCase();
```

configura `bigRiver` como o objeto `String` `"MISSISSIPPI"`.

Ao aplicar um método a um objeto, você deve certificar-se de que esse método esteja definido na classe apropriada. Por exemplo, é um erro chamar

```
System.out.length(); // Essa chamada de método é um erro.
```

A classe `PrintStream` (à qual `System.out` pertence) não possui um método `length`.

A interface pública de uma classe especifica o que você pode fazer com os objetos dela. A implementação oculta descreve como essas ações são executadas.

Vamos resumir. Em Java, *cada objeto pertence a uma classe. A classe define os métodos para os objetos.* Por exemplo, a classe `String` define os métodos `length` e `toUpperCase` (bem como outros métodos – veremos a maioria deles no Capítulo 4). Os métodos formam a interface *pública* da classe e determinam o que você pode fazer com os objetos dela. Uma classe também define uma *implementação privada*, que descreve os dados dentro dos seus objetos e as instruções para seus métodos. Esses detalhes permanecem ocultos dos programadores, os quais utilizam objetos e métodos de chamada.

A Figura 4 mostra dois objetos da classe `String`. Cada objeto armazena seus próprios dados (desenhados como caixas que contêm caracteres). Ambos suportam o mesmo conjunto de métodos – a interface que é especificada pela classe `String`.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

6. Como você pode calcular o comprimento da string `"Mississippi"`?
7. Como você pode imprimir a versão em letras maiúsculas de `"Hello, World!"`?
8. É válido chamar `river.println()`? Por que sim ou por que não?

2.4 Parâmetros de método e valores de retorno

Nesta seção, examinaremos como fornecer entradas em um método e como obter a saída do método.

Um parâmetro é uma entrada para um método.

Alguns métodos requerem entradas que fornecem detalhes sobre o trabalho que precisam fazer. Por exemplo, o método `println` tem uma entrada: a string que deve ser impressa. Cientistas da computação utilizam o termo técnico *parâmetro* para entradas de método. Dizemos que a string `greeting` é um parâmetro da chamada de método

```
System.out.println(greeting)
```

A Figura 5 ilustra a passagem do parâmetro para o método.

O parâmetro implícito de uma chamada de método é o objeto em que o método é invocado.

Tecnicamente falando, o parâmetro `greeting` é um *parâmetro explícito* do método `println`. O objeto em que você invoca o método também é considerado um parâmetro da chamada de método, e é denominado *parâmetro implícito*. Por exemplo, `System.out` é o parâmetro implícito da seguinte chamada de método:

```
System.out.println(greeting)
```

Alguns métodos requerem múltiplos parâmetros explícitos, outros não requerem absolutamente nenhum. Um exemplo do último é o método `length` da classe `String` (veja Figura 6). Todas as informações que o método `length` requer para fazer o trabalho – a saber, a seqüência de caracteres da string – estão armazenadas no próprio objeto parâmetro implícito.

O valor de retorno de um método é o resultado que o método calculou para uso pelo código que o chamou.

O método `length` difere do método `println` de uma outra maneira: ele tem uma saída. Dizemos que o método *retorna um valor*, a saber, o número de caracteres na string. Você pode armazenar o valor de retorno em uma variável:

```
int n = greeting.length();
```

Você também pode utilizar o valor de retorno como um parâmetro de outro método:

```
System.out.println(greeting.length());
```

A chamada de método `greeting.length()` retorna um valor – o inteiro 13. O valor de retorno torna-se um parâmetro do método `println`. A Figura 7 mostra o processo.

Nem todos os métodos retornam valores. Um exemplo é o método `println`. O método `println` interage com o sistema operacional, fazendo com que os caracteres apareçam em uma janela. Mas ele não retorna um valor ao código que o chama.

Vamos analisar uma chamada de método mais complexa. Aqui, chamaremos o método `replace` da classe `String`. O método `replace` executa uma operação de pesquisa e substituição, semelhante àquela de um processador de texto. Por exemplo, a chamada:

```
river.replace("issipp", "our")
```

constrói uma nova string que é obtida substituindo todas as ocorrências de `"issipp"` em `"Mississippi"` por `"our"`. (Nessa situação, há somente uma substituição.) O método retorna o objeto `String` `"Missouri"` (que tanto pode ser salvo em uma variável como passado para outro método).

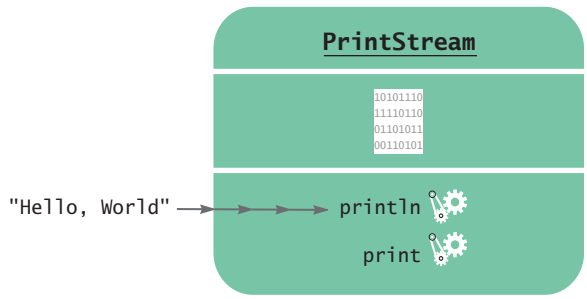


Figura 5 Passando um parâmetro para o método `println`.

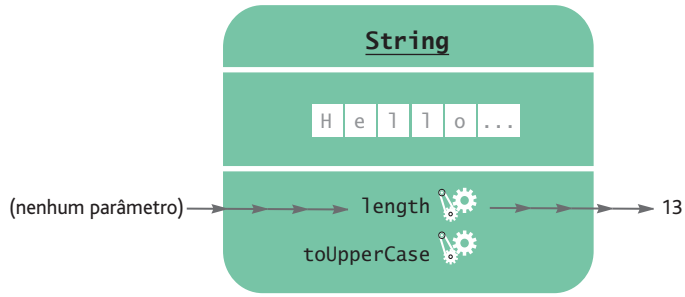


Figura 6 Invocando o método `length` em um objeto `String`.

Como a Figura 8 mostra, essa chamada de método tem:

- um parâmetro implícito: a string `"Mississippi"`
- dois parâmetros explícitos: as strings `"issipp"` e `"our"`
- um valor de retorno: a string `"Missouri"`

Quando um método é definido em uma classe, essa definição especifica os tipos dos parâmetros explícitos e o valor de retorno. Por exemplo, a classe `String` define o método `length` como:

```
public int length()
```

Isto é, não há parâmetro explícito e o valor de retorno é do tipo `int`. (Por enquanto, todos os métodos que consideramos serão métodos "públicos" – ver o Capítulo 10 para métodos mais restritos.)

O tipo do parâmetro implícito é a classe que define o método – `String` no nosso caso. Ele não é mencionado na definição de método – daí o termo "implícito".

O método `replace` é definido como

```
public String replace(String target, String replacement)
```

Para chamar o método `replace`, você fornece dois parâmetros explícitos, `target` e `replacement`, que são do tipo `String`. O valor retornado é uma outra string.

Quando um método não retorna valor algum, o tipo de retorno é declarado com a palavra reservada `void`. Por exemplo, a classe `PrintStream` define o método `println` como:

```
public void println(String output)
```

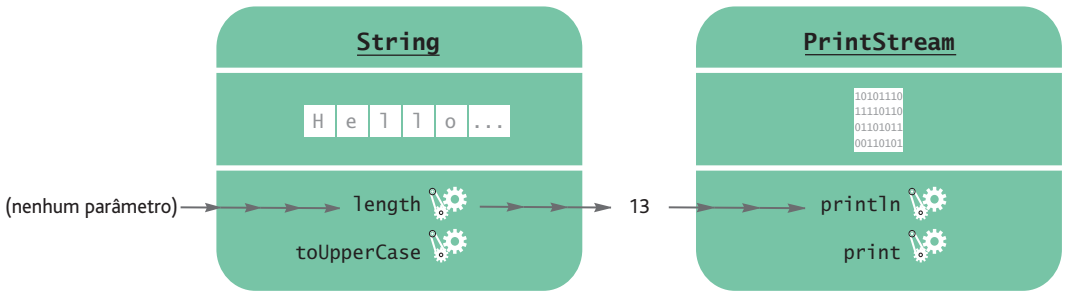


Figura 7 Passando o resultado de uma chamada de método para outro método.

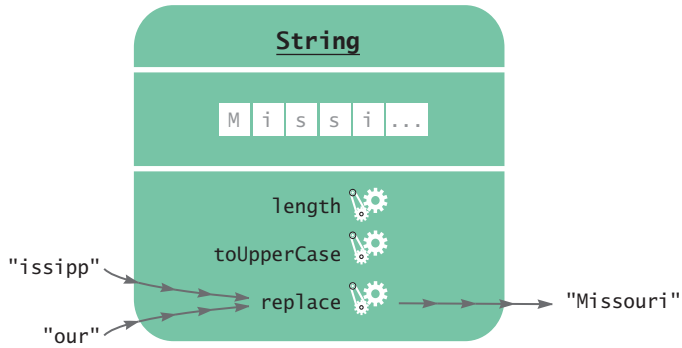


Figura 8 Chamando o método `replace`.

Um nome de método é sobrecarregado se uma classe tiver mais de um método com o mesmo nome (mas tipos diferentes de parâmetros).

Ocasionalmente, uma classe define dois métodos com o mesmo nome e diferentes tipos de parâmetros explícitos. Por exemplo, a classe `PrintStream` define um segundo método, também chamado `println`, como

```
public void println(int output)
```

Esse método é utilizado para imprimir um valor inteiro. Dizemos que o nome `println` é *sobrecarregado* porque referencia mais de um método.

Autoverificação da aprendizagem

9. Quais são os parâmetros implícitos, os parâmetros explícitos e os valores de retorno na chamada do método `river.length()`?
10. Qual é o resultado da chamada `river.replace("p", "s")`?
11. Qual é o resultado da chamada `greeting.replace("World", "Dave").length()`?
12. Como o método `toUpperCase` é definido na classe `String`?

2.5 Tipos numéricos

O tipo `double` indica números de ponto flutuante que podem ter partes fracionárias.

Java possui tipos separados para *inteiros* e *números de ponto flutuante*. Inteiros são números sem parte fracionária; números de ponto flutuante podem ter partes fracionárias. Por exemplo, 13 é um inteiro e 1.3* é um número de ponto flutuante.

O nome "ponto flutuante" descreve a representação do número no computador como uma seqüência dos dígitos significativos e uma indicação da posição do ponto decimal. Por exemplo, os números 13000, 1.3, 0.00013 têm os mesmos dígitos decimais: 13. Quando um número de ponto flutuante é multiplicado ou dividido por 10, somente a posição do ponto da fração decimal muda; ele "flutua". Essa representação

* N. de T.: Em Java, a vírgula decimal é representada por um ponto. Da mesma forma, o ponto utilizado para dividir as centenas é substituído por uma vírgula, seguindo a notação de representação de números utilizada no hemisfério norte. É importante que o leitor não se confunda.

está relacionada à notação científica 1.3×10^{-4}. (Na verdade, o computador representa números na base 2, não na base 10, mas o princípio é o mesmo.)

Se precisar processar números com uma parte fracionária, você deverá utilizar o tipo chamado `double`, que significa "número de ponto flutuante de dupla precisão". Pense em um número no formato `double` como qualquer número que pode aparecer no painel de exibição de uma calculadora, como 1.3 ou –0.333333333.

Não utilize pontos para separar os milhares ao escrever números em Java. Por exemplo, 13.000 deve ser escrito como `13000`. Para escrever números na notação exponencial em Java, utilize a notação `E`*n* em vez de "$\times 10^n$". Por exemplo, 1.3×10^{-4} é escrito como `1.3E-4`.

Você poderia perguntar por que Java tem tipos diferentes para números inteiros e números de ponto flutuante. Calculadoras de bolso não precisam de um tipo inteiro separado; elas utilizam números de ponto flutuante para todos os cálculos. Mas inteiros têm várias vantagens em relação aos números de ponto flutuante. Eles ocupam menos espaço de armazenamento, são processados mais rapidamente e não provocam erros de arredondamento. É recomendável utilizar os tipos `int` para quantidades que nunca podem ter partes fracionárias, como o comprimento de uma string. Utilize os tipos `double` para representar quantidades que podem ter partes fracionárias, como a média de uma nota.

Há vários outros tipos numéricos em Java que não são comumente utilizados. Discutiremos esses tipos no Capítulo 4. Mas, para propósitos mais práticos, os tipos `int` e `double` são tudo o que você precisa para processar números.

Em Java, números não são objetos e tipos numéricos não são classes.

Em Java, os tipos numéricos (`int`, `double` e tipos menos comumente utilizados) são *tipos primitivos*, não classes. Números não são objetos. Assim, os tipos numéricos não têm método.

Os números podem ser combinados por operadores aritméticos como +, – e *.

Mas você pode combinar números a operadores, por exemplo, `+` e `-`, como em `10 + n` ou `n - 1`. Para multiplicar dois números, utilize o operador `*`. Por exemplo, $10 \times n$ é escrito como `10 * n`.

Como na matemática, o operador `*` tem precedência sobre o operador `+`. Isto é, `x + y * 2` significa a soma de `x` com `y * 2`. Se desejar multiplicar a soma de `x` e `y` por 2, use parênteses:

```
(x + y) * 2
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

13. Qual tipo de número você utilizaria para armazenar a área de um círculo?
14. Por que a expressão `13.println()` é um erro?
15. Escreva uma expressão para calcular a média dos valores `x` e `y`.

2.6 Construindo objetos

A maioria dos programas Java trabalha com vários objetos. Nesta seção, você verá como *construir* novos objetos. Ela permite ir além dos objetos `String` e do objeto `System.out` predefinido.

Para entender a construção de um objeto, vamos passar para uma outra classe: a classe `Rectangle` na biblioteca de classes Java. Os objetos do tipo `Rectangle` descrevem

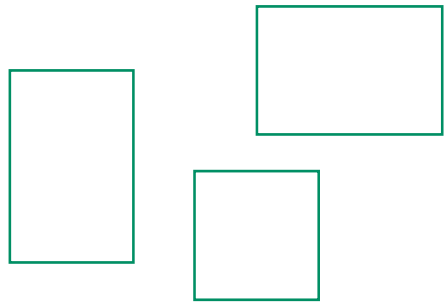


Figura 9 Formas retangulares.

formas retangulares – veja Figura 9. Esses objetos são úteis para vários propósitos. Você pode montar retângulos em gráficos de barras e programar jogos simples movendo os retângulos para dentro de uma janela.

Observe que um objeto `Rectangle` não é uma forma retangular – ele é um objeto que contém um conjunto de números. Os números *descrevem* o retângulo (veja Figura 10). Cada retângulo é descrito pelas coordenadas x e y do seu canto superior esquerdo, sua largura e sua altura.

É muito importante que você entenda essa distinção. No computador, um objeto `Rectangle` é um bloco de memória que armazena quatro números, por exemplo, $x = 5$, $y = 10$, *largura* = 20, *altura* = 30. Na imaginação do programador que usa um objeto `Rectangle`, esse objeto descreve uma figura geométrica.

Utilize o operador `new`, seguido por um nome de classe e parâmetros para construir novos objetos.

Para criar um novo retângulo, você precisa especificar os valores de x, y, *largura* e *altura*. Então *invoque o* operador `new`, especificando o nome da classe e os parâmetros necessários para construir um novo objeto. Por exemplo, você pode criar um novo retângulo com o canto superior esquerdo em (5, 10), largura 20 e altura 30 como a seguir:

```
new Rectangle(5, 10, 20, 30)
```

Eis o que acontece em detalhe.

1. O operador `new` cria um objeto `Rectangle`.
2. Usa os parâmetros (nesse caso, 5, 10, 20 e 30) para inicializar os dados do objeto.
3. Retorna o objeto.

Normalmente, a saída do operador `new` é armazenada em uma variável. Por exemplo,

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

O processo da criação de um novo objeto é chamado *construção*. Os quatro valores 5, 10, 20 e 30 são chamados *parâmetros de construção*. Observe que a expressão `new` *não* é

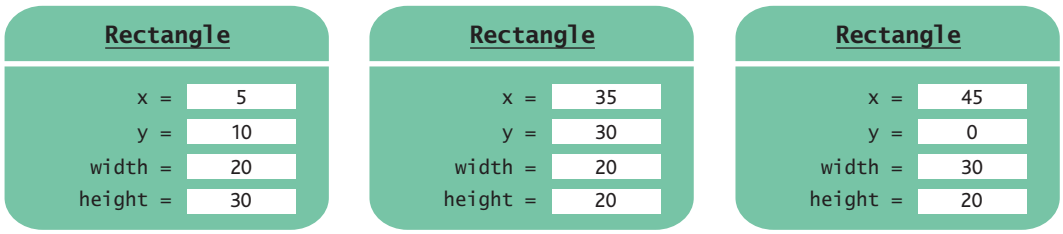


Figura 10 Objetos `Rectangle`.

uma instrução completa. Você usa o valor de uma expressão `new` da mesma maneira que um valor de retorno de método: Atribua-o a uma variável ou passe-o para outro método.

Algumas classes permitem construir objetos de várias maneiras. Por exemplo, você também pode obter um objeto `Rectangle` sem fornecer um parâmetro de construção (mas você ainda deve fornecer os parênteses):

```
new Rectangle()
```

Essa expressão constrói um retângulo (bastante inútil) com o canto superior esquerdo na origem (0, 0), largura 0 e altura 0.

SINTAXE 2.3 Construção de objeto

`new` *NomeDaClasse*(*parâmetros*)

Exemplo:

```
new Rectangle(5, 10, 20, 30)
new Rectangle()
```

Objetivo:

Construir um novo objeto. Inicialize-o com os parâmetros de construção e retorne uma referência ao objeto construído

AUTOVERIFICAÇÃO DA APRENDIZAGEM

16. Como você constrói um quadrado com centro (100, 100) e comprimento de lado 20?
17. O que a instrução a seguir imprime?

```
System.out.println(new Rectangle().getWidth());
```

ERRO COMUM 2.1

Tentando invocar um construtor como se fosse um método

Construtores não são métodos. Você só pode utilizar um construtor com o operador `new`, não para reinicializar um objeto existente:

```
box.Rectangle(20, 35, 20, 30); // Erro – não é possível reinicializar o objeto.
```

A correção é simples: Crie um novo objeto e sobrescreva o atual.

```
box = new Rectangle(20, 35, 20, 30); // OK
```

2.7 Métodos de acesso e métodos modificadores

Um método de acesso não altera o estado do seu parâmetro implícito. Um método modificador altera o estado.

Nesta seção, introduziremos uma terminologia útil para os métodos de uma classe. Um método que acessa um objeto e retorna algumas informações sobre ele, sem alterá-lo, é chamado de *método de acesso*. Em comparação, um método cujo propósito é modificar o estado de um objeto é chamado de método *modificador*.

Por exemplo, o método `length` da classe `String` é um método de acesso. Ele retorna informações sobre uma string, a saber, seu comprimento. Mas ele não modifica a string ao contar os caracteres.

A classe `Rectangle` tem alguns métodos de acesso. Os métodos `getX`, `getY`, `getWidth` e `getHeight` retornam as coordenadas *x* e *y* do canto superior esquerdo e os valores de largura e altura. Por exemplo,

```
double width = box.getWidth();
```

Agora vamos considerar um método modificador. Programas que manipulam retângulos freqüentemente precisam movimentá-los para, por exemplo, exibir animações. A classe `Rectangle` tem um método para esse propósito, chamado `translate`. (Matemáticos utilizam o termo "translação" para um movimento rígido do plano.) Esse método move um retângulo por certa distância nas direções *x* e *y*. A chamada de método:

```
box.translate(15, 25);
```

move o retângulo por 15 unidades na direção *x* e 25 unidades na direção *y* (veja Figura 11). Mover um retângulo não muda a largura nem a altura, mas altera o canto superior esquerdo. Após o movimento, o canto superior esquerdo estará em (20, 35).

Esse é um método modificador porque altera o objeto parâmetro implícito.

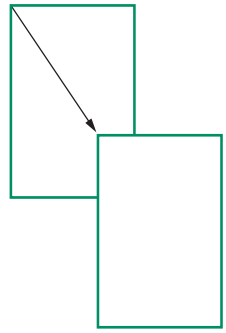


Figura 11
Utilizando o método `translate` para mover um retângulo.

Autoverificação da aprendizagem

18. O método `toUpperCase` da classe `String` é um método de acesso ou um modificador?
19. Que chamada ao método `translate` é necessária para mover o retângulo `box` de modo que seu canto superior esquerdo seja a origem (0, 0)?

2.8 Implementando um programa de teste

Nesta seção, discutiremos os passos necessários para implementar um programa de teste. O propósito de um programa de teste é verificar se um ou mais métodos foram implementados corretamente. Um programa de teste chama métodos e verifica se eles retornam os resultados esperados. Escrever programas de teste é uma atividade muito importante. Ao implementar seus próprios métodos, você sempre deve fornecer programas para testá-los.

Neste livro, utilizamos um formato muito simples para programas de teste. Você verá agora um desses programas que testa um método na classe `Rectangle`. Esse programa realiza as seguintes ações:

1. Fornece uma classe testadora.
2. Provê um método `main`.
3. Dentro do método `main`, constrói um ou mais objetos.
4. Aplica métodos aos objetos.
5. Exibe os resultados das chamadas de método.
6. Exibe os valores que você espera obter.

Você precisa seguir esses passos sempre que escrever um programa para testar suas classes.

Nosso programa de teste de exemplo testa o comportamento do método `translate`. Eis os principais passos (que foram colocados dentro do método `main` da classe `RectangleTester`).

```
Rectangle box = new Rectangle(5, 10, 20, 30);

// Move o retângulo
box.translate(15, 25);

// Imprime informações sobre o retângulo movido
System.out.print("x: ");
System.out.println(box.getX());
System.out.println("Expected: 20");
```

Imprimimos o valor retornado pelo método `getX` e então imprimimos uma mensagem que descreve o valor que esperamos ver.

Determinar o resultado esperado antecipadamente é uma parte importante do processo de teste.

Esse é um passo muito importante. Pare um pouco para pensar qual é o resultado esperado antes de você executar um programa de teste. Essa consideração prévia do processo irá ajudá-lo a entender como seu programa deve se comportar e pode ajudá-lo a rastrear erros no começo do desenvolvimento de um projeto.

No nosso caso, o retângulo foi construído com o canto esquerdo superior em (5, 10). A direção x foi movida por 15 pixels, portanto esperamos um valor x de $5 + 15 = 20$ depois dessa movimentação.

Eis um programa completo que testa a movimentação de um retângulo.

ch02/rectangle/MoveTester.java

```
1   import java.awt.Rectangle;
2
3   public class MoveTester
4   {
5      public static void main(String[] args)
6      {
7         Rectangle box = new Rectangle(5, 10, 20, 30);
8
9         // Move o retângulo
10        box.translate(15, 25);
11
```

```
12        // Imprime informações sobre o retângulo movido
13        System.out.print("x: ");
14        System.out.println(box.getX());
15        System.out.println("Expected: 20");
16
17        System.out.print("y: ");
18        System.out.println(box.getY());
19        System.out.println("Expected: 35");
20     }
21  }
```

Saída

```
x: 20
Expected: 20
y: 35
Expected: 35
```

Classes Java são agrupadas em pacotes. Utilize a instrução `import` para usar classes definidas em outros pacotes.

Para esse programa, precisamos seguir outro passo: Precisamos *importar* a classe `Rectangle` a partir de um *pacote*. Um pacote é uma coleção de classes com um propósito relacionado. Todas as classes na biblioteca padrão estão contidas em pacotes. A classe `Rectangle` pertence ao pacote `java.awt` (`awt` é uma abreviação para "Abstract Windowing Toolkit"), que contém muitas classes para desenhar janelas e formas gráficas.

Para utilizar a classe `Rectangle` no pacote `java.awt`, simplesmente posicione a linha a seguir na parte superior do seu programa:

```
import java.awt.Rectangle;
```

Por que não precisa importar as classes `System` e `String`? Porque as classes `System` e `String` estão no pacote `java.lang`, e todas as classes nesse pacote são automaticamente importadas, assim você nunca precisará importá-las.

SINTAXE 2.4 Importando uma classe a partir de um pacote

`import` *nomeDoPacote*`.`*NomeDaClasse*`;`

Exemplo:

```
import java.awt.Rectangle;
```

Objetivo:

Importar uma classe a partir de um pacote para uso em um programa

AUTOVERIFICAÇÃO DA APRENDIZAGEM

20. Suponha que `box.translate(25, 15)` tenha sido chamada em vez de `box.translate(15, 25)`. Quais são as saídas esperadas?

21. Por que o programa `MoveTester` não precisa imprimir a largura e a altura do retângulo?
22. A classe `Random` é definida no pacote `java.util`. O que você precisa fazer para utilizar essa classe no seu programa?

TÓPICO AVANÇADO 2.1

Testando classes em um ambiente interativo

O Tópico Avançado 2.1 descreve como classes podem ser testadas facilmente no ambiente BlueJ, sem ter de escrever uma classe testadora separada.

2.9 A documentação da API

As classes e métodos da biblioteca Java estão listados na *documentação da API*. A API (*application programming interface*) é a interface de programação de aplicativo. Um programador que usa as classes Java para montar um programa de computador (ou *aplicativo*) é um *programador de aplicativo*. Esse é você. Em comparação, os programadores que projetaram e implementaram as classes de biblioteca como `PrintStream` e `Rectangle` são *programadores de sistema*.

A documentação da API (*Application Programming Interface*) lista as classes e métodos da biblioteca Java.

Você pode encontrar a documentação da API na Web [1] (`http://java.sun.com/javase/6/docs/api/index.html`). Alternativamente, você pode fazer download e instalar a documentação da API em seu próprio computador – veja a Dica de produtividade 2.1.

A documentação da API documenta todas as classes na biblioteca Java – há milhares delas (veja Figura 12). A maioria das classes é bastante especializada e somente algumas têm interesse ao programador iniciante.

Localize o link `Rectangle` no painel esquerdo, preferencialmente utilizando a função de pesquisa do seu navegador. Clique nesse link e o painel direito mostrará todos os recursos da classe `Rectangle` (veja Figura 13).

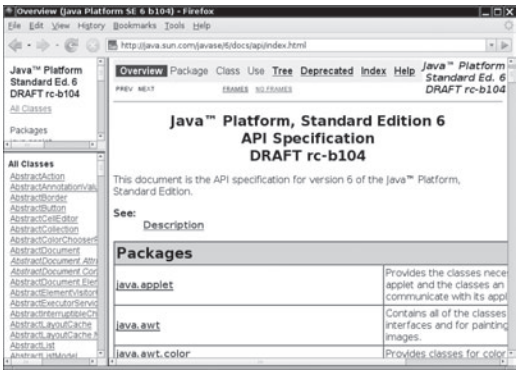


Figura 12 Documentação da API para biblioteca Java padrão.

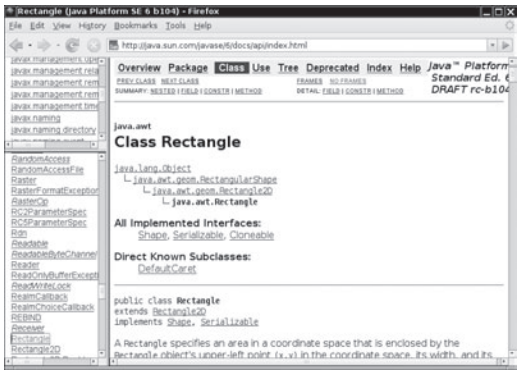


Figura 13 Documentação da API para a classe `Rectangle`.

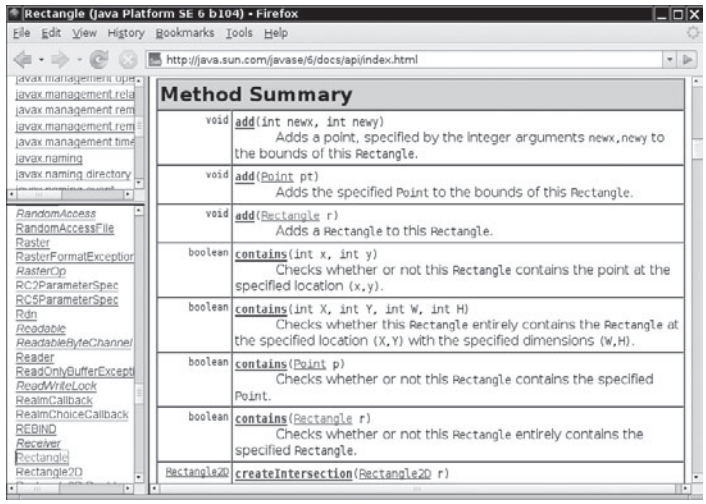


Figura 14 Resumo dos métodos da classe `Rectangle`.

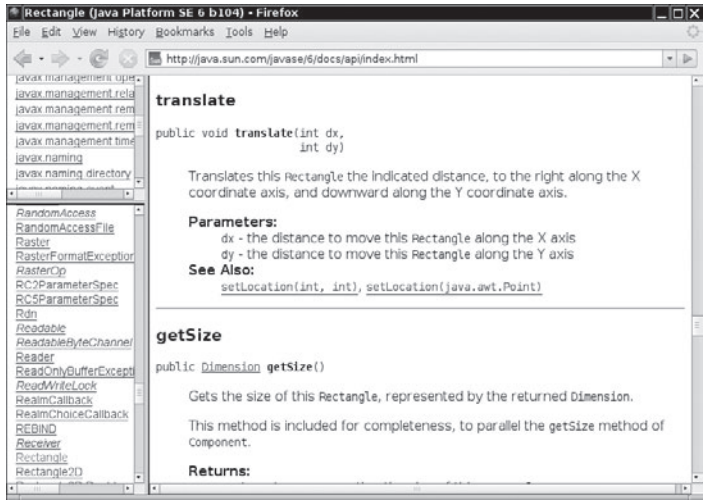


Figura 15 Documentação da API para método `translate`.

A documentação da API para cada classe inicia com uma seção que descreve o propósito da classe. Em seguida, vêm as tabelas de resumo para os construtores e métodos (veja Figura 14). Clique no link de um método para obter uma descrição detalhada (veja Figura 15).

Como você pode ver, a classe `Rectangle` tem muitos métodos. Embora ela possa intimidar o programador iniciante algumas vezes, essa é umas das forças da biblioteca padrão. Se um dia você precisar fazer um cálculo que envolve retângulos, provavelmente haverá um método que faz todo o trabalho para você.

O Apêndice C contém uma versão resumida da documentação da API. Talvez você ache que a documentação resumida seja mais fácil de utilizar do que a documentação

integral. Tudo bem se você contar com a documentação resumida para seus primeiros programas, mas com o tempo você deverá passar para a coisa real.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

23. Examine a documentação da API para a classe `String`. Qual método você utilizaria para obter a string `"hello, world!"` a partir da string `"Hello, World!"`?
24. Na documentação da API para a classe `String`, examine a descrição do método `trim`. Qual é o resultado da aplicação de `trim` à string `" Hello, Space ! "`? (Observe os espaços na string.)

DICA DE PRODUTIVIDADE 2.1

Não memorize – use a ajuda on-line

A biblioteca Java tem milhares de classes e métodos. Não é necessário nem útil tentar memorizá-los. Em vez disso, você deve familiarizar-se com o uso da documentação da API. Uma vez que você precisará utilizar a documentação da API todo o tempo, é melhor baixá-la e instalá-la no seu computador, particularmente se você não tem uma conexão permanente com a Internet. Você pode baixar a documentação em `http://java.sun.com/javase/downloads/index.html`.

2.10 Referências a objetos

Em Java, uma variável cujo tipo é uma classe, na verdade não contém um objeto. Ela contém meramente a *posição de um objeto na memória*. O objeto em si é armazenado em uma outra parte – veja Figura 16.

Uma referência a objeto descreve a localização de um objeto.

Utilizamos o termo técnico *referência a objeto* significando a posição de um objeto na memória. Quando uma variável contém a posição de um objeto na memória, dizemos que ela *referencia* um objeto. Por exemplo, depois da instrução

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

a variável `box` referencia o objeto `Rectangle` que o operador `new` construiu. Tecnicamente falando, o operador `new` retornou uma referência ao novo objeto e essa referência é armazenada na variável `box`.

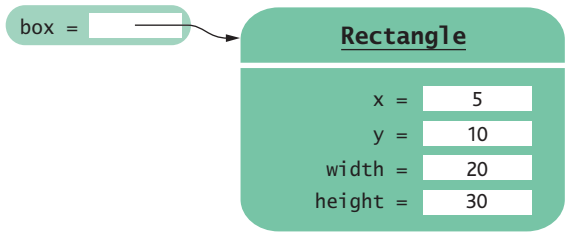


Figura 16 Uma variável de objeto contendo uma referência a objeto.

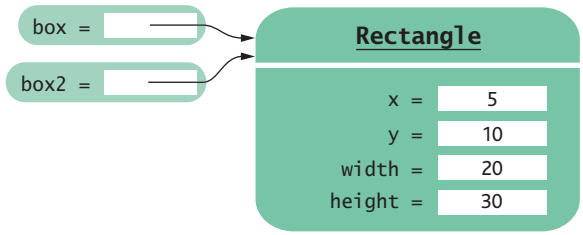


Figura 17 Duas variáveis de objeto referenciando o mesmo objeto.

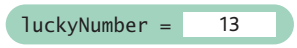


Figura 18 Uma variável numérica armazena um número.

É muito importante lembrar-se de que a variável `box` *não contém* o objeto. Ela *referencia* o objeto. Você pode fazer duas variáveis de objeto referenciarem o mesmo objeto:

```
Rectangle box2 = box;
```

> Múltiplas variáveis de objeto podem conter referências ao mesmo objeto.

Agora você pode acessar o mesmo objeto `Rectangle` como `box` ou como `box2`, conforme mostrado na Figura 17.

Mas variáveis numéricas, na verdade, armazenam números. Quando você define

```
int luckyNumber = 13;
```

a variável `luckyNumber` contém o número 13, não uma referência ao número (veja Figura 18).

> Variáveis numéricas armazenam números. Variáveis de objeto armazenam referências.

Você pode ver a diferença entre variáveis numéricas e variáveis de objeto ao criar uma cópia de uma variável. Ao copiar um valor de tipo primitivo, o número original e a cópia do número são valores independentes. Mas quando você copia uma referência a objeto, tanto o original como a cópia são referências ao mesmo objeto.

Pense no código a seguir, que copia um número e então altera a cópia (veja Figura 19):

```
int luckyNumber = 13; ①
int luckyNumber2 = luckyNumber; ②
luckyNumber2 = 12; ③
```

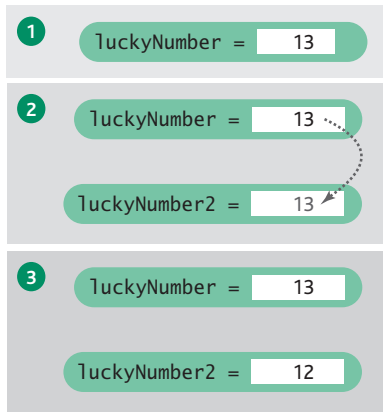


Figura 19 Copiando números.

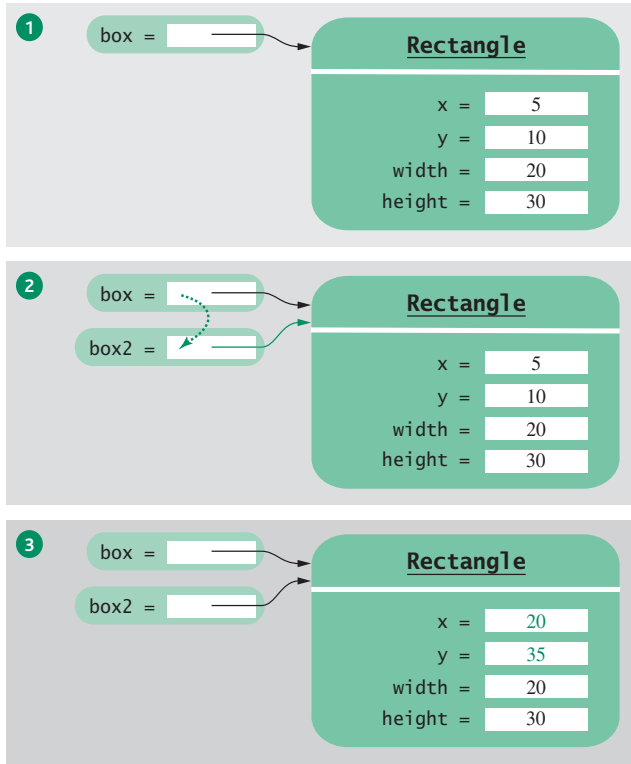


Figura 20 Copiando referências a objeto.

Agora a variável `luckyNumber` contém o valor 13 e `luckyNumber2` contém 12. Agora considere o código aparentemente análogo com objetos `Rectangle`.

```
Rectangle box = new Rectangle(5, 10, 20, 30); 1
Rectangle box2 = box; // See Figure 20 2
box2.translate(15, 25); 3
```

Como `box` e `box2` referem-se ao mesmo retângulo depois do passo 2, ambas as variáveis referenciam o retângulo movido após a chamada ao método `translate`.

Há uma razão para essa diferença entre números e objetos. No computador, cada número requer uma pequena quantidade de memória para seu armazenamento. Mas os objetos podem ser muito grandes. É muito mais eficiente manipular somente a sua posição na memória.

Francamente falando, a maioria dos programadores não se preocupa muito com a diferença entre objetos e referências a objeto. Boa parte do tempo, você terá a intuição correta ao pensar no "objeto `box`" em vez de "a referência ao objeto armazenada em `box`", tecnicamente mais exata. A diferença entre objetos e referências a objeto só é aparente quando você tem múltiplas variáveis que se referem ao mesmo objeto.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

25. Qual é o efeito da atribuição `greeting2 = greeting`?

26. Depois de chamar `greeting2.toUpperCase()`, qual é o conteúdo de `greeting` e `greeting2`?

FATO ALEATÓRIO 2.1

Mainframes – quando os dinossauros dominavam a Terra

O Fato Aleatório 2.1 descreve a história dos computadores mainframe, computadores extremamente caros que ocupavam salas inteiras e eram a base da computação entre 1950 e 1980.

2.11 Aplicações gráficas e janelas de frame

Esta é a primeira de várias seções que ensinam a escrever *aplicações gráficas*: aplicações que exibem desenhos dentro de uma janela. Aplicações gráficas são mais atraentes do que as aplicações de console que exibem texto simples em uma janela de console.

O material nesta seção, bem como nas seções denominadas "Exercícios gráficos" nos outros capítulos, são inteiramente opcionais. Sinta-se livre para pulá-las se você não estiver interessado em desenhar elementos gráficos.

Para mostrar um frame, construa um objeto `JFrame`, configure o tamanho e torne-o visível.

Uma aplicação gráfica mostra as informações dentro de uma janela de frame: uma janela com uma moldura e uma barra de título, como mostrado na Figura 21. Nesta seção, você aprenderá a exibir uma janela de frame. Na Seção 3.9, você aprenderá a criar um desenho dentro do frame.

Para mostrar um frame, siga estes passos:

1. Construa um objeto da classe `JFrame`:

```
JFrame frame = new JFrame();
```

2. Configure o tamanho do frame.

```
frame.setSize(300, 400);
```

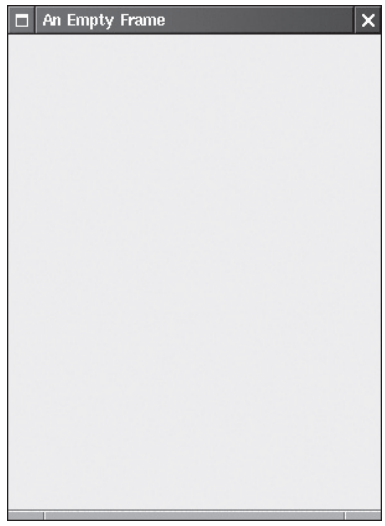


Figura 21 Uma janela de frame.

Esse frame terá 300 pixels de largura e 400 pixels de altura. Se omitir esse passo o frame terá 0 por 0 pixels e você não será capaz de vê-lo.

3. Se quiser, configure o título do frame.

```
frame.setTitle("An Empty Frame");
```

Se omitir esse passo, a barra de título será simplesmente deixada em branco.

4. Configure a "operação padrão de fechamento":

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Quando o usuário fecha o frame, o programa é automaticamente fechado. Não omita esse passo. Se omitir, o programa continuará em execução mesmo depois de o frame ser fechado.

5. Torne o frame visível.

```
frame.setVisible(true);
```

O programa simples abaixo mostra todos esses passos. Ele produz o frame vazio mostrado na Figura 21.

A classe `JFrame` é parte do pacote `javax.swing`. Swing é o apelido para a biblioteca de interface gráfica com o usuário de Java. O "x" em `javax` denota o fato de que o Swing inicialmente foi projetado como uma *extensão* do Java antes de ser adicionado à biblioteca padrão.

Examinaremos a programação com Swing mais detalhadamente nos Capítulos 3, 9, 10 e 18. Por enquanto, considere este programa como o código básico essencial necessário para mostrar um frame.

ch02/emptyframe/EmptyFrameViewer.java

```
1  import javax.swing.JFrame;
2
3  public class EmptyFrameViewer
4  {
5     public static void main(String[] args)
6     {
7        JFrame frame = new JFrame();
8
9        frame.setSize(300, 400);
10       frame.setTitle("An Empty Frame");
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12
13       frame.setVisible(true);
14    }
15 }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

27. Como você exibe um frame quadrado com uma barra de título na qual se lê "Hello, World!"?
28. Como um programa pode exibir dois frames de uma só vez?

2.12 Desenhando em um componente

Esta seção continua a seção opcional de exercícios gráficos. Você aprenderá a fazer formas aparecerem dentro de uma janela de frame. O primeiro desenho será bem modesto: apenas dois retângulos (veja Figura 22). Logo você verá como produzir desenhos mais interessantes. O propósito desse exemplo é mostrar o princípio básico de um programa que cria um desenho. Você não pode desenhar diretamente sobre um frame. Sempre que quiser exibir alguma coisa dentro de um frame, seja um botão ou um desenho, você terá de construir um objeto *componente* e adicioná-lo ao frame. No conjunto de ferramentas Swing, a classe `JComponent` representa um componente em branco.

Para exibir um desenho em um frame, defina uma classe que estende a classe `JComponent`.

Como não queremos adicionar um componente em branco, temos de modificar a classe `JComponent` e especificar a maneira como o componente deve ser pintado. A solução é definir uma nova classe que estende a classe `JComponent`. Você aprenderá a estender classes no Capítulo 10. Por enquanto, simplesmente utilize o código a seguir como um modelo.

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      As instruções de desenho entram aqui
   }
}
```

Coloque as instruções de desenho dentro do método `paintComponent`. Esse método é chamado sempre que o componente precisa ser desenhado.

A palavra-chave `extends` indica que nossa classe de componentes, `RectangleComponent`, herda os métodos de `JComponent`. Mas a classe `RectangleComponent` é diferente da classe `JComponent` básica em um detalhe: O método `paintComponent` conterá instruções para desenhar os retângulos.

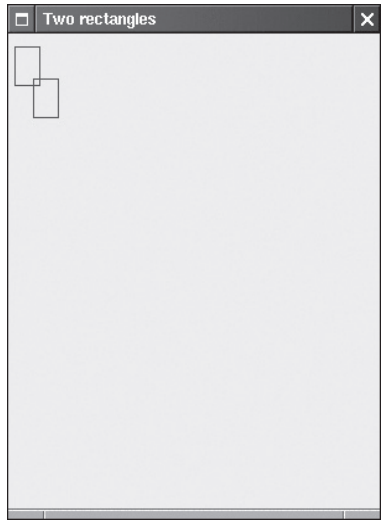


Figura 22 Desenhando retângulos.

Quando a janela é mostrada pela primeira vez, o método `paintComponent` é chamado automaticamente. Esse método também é chamado quando a janela é redimensionada ou quando é mostrada novamente depois de ter sido escondida.

A classe `Graphics` permite manipular o estado dos elementos gráficos (por exemplo, as cores atuais).

O método `paintComponent` recebe um objeto do tipo `Graphics`. O objeto `Graphics` armazena o estado atual dos elementos gráficos – cor atual, fonte, etc., que são utilizados nas operações de desenho.

Use coerção (*typecasting*) para recuperar o objeto `Graphics2D` a partir do parâmetro `Graphics` do método `paintComponent`.

Entretanto, a classe `Graphics` é primitiva. Quando programadores solicitaram uma abordagem mais orientada a objetos para desenhar elementos gráficos, os projetistas de Java criaram a classe `Graphics2D`, que estende a classe `Graphics`. Sempre que o conjunto de ferramentas Swing chama o método `paintComponent`, na realidade ele passa um parâmetro do tipo `Graphics2D`. Programas com elementos gráficos simples não precisam desse recurso. Como queremos utilizar os métodos mais sofisticados para desenhar objetos gráficos bidimensionais, precisamos recuperar a classe `Graphics2D`. Isso é feito utilizando *coerção* (ou *typecasting*):

A classe `Graphics2D` possui métodos para desenhar formas.

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Recupera Graphics2D
      Graphics2D g2 = (Graphics2D) g;
      . . .
   }
}
```

Agora você está pronto para desenhar formas. O método `draw` da classe `Graphics2D` pode desenhar formas, por exemplo, retângulos, elipses, segmentos de linha, polígonos e arcos. Aqui desenharemos um retângulo:

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      . . .
      Rectangle box = new Rectangle(5, 10, 20, 30);
      g2.draw(box);
      . . .
   }
}
```

Em seguida apresentamos o código-fonte para a classe `RectangleComponent`. Observe que o método `paintComponent` da classe `RectangleComponent` desenha dois retângulos.

Como você pode ver a partir das instruções `import`, as classes `Graphics` e `Graphics2D` são parte do pacote `java.awt`.

ch02/rectangles/RectangleComponent.java

```
1  import java.awt.Graphics;
2  import java.awt.Graphics2D;
3  import java.awt.Rectangle;
```

```
4  import javax.swing.JComponent;
5
6  /**
7     Um componente que desenha dois retângulos.
8  */
9  public class RectangleComponent extends JComponent
10 {
11    public void paintComponent(Graphics g)
12    {
13       // Recupera Graphics2D
14       Graphics2D g2 = (Graphics2D) g;
15
16       // Constrói um retângulo e o desenha
17       Rectangle box = new Rectangle(5, 10, 20, 30);
18       g2.draw(box);
19
20       // Move o retângulo 15 unidades para a direita e 25 unidades para baixo
21       box.translate(15, 25);
22
23       // Desenha o retângulo movido
24       g2.draw(box);
25    }
26 }
```

Para ver o desenho, é preciso mais uma tarefa. Você precisa exibir o frame em que você adicionou um objeto componente. Siga estes passos:

1. Construa um frame como descrito na seção anterior.
2. Construa um objeto da sua classe de componentes:

   ```
   RectangleComponent component = new RectangleComponent();
   ```

3. Adicione o componente ao frame:

   ```
   frame.add(component);
   ```

4. Torne o frame visível, como descrito na seção anterior.

A listagem a seguir mostra o processo completo.

ch02/rectangles/RectangleViewer.java

```
1  import javax.swing.JFrame;
2
3  public class RectangleViewer
4  {
5  pubic static void main(String)[] args)
6  {
7        JFrame frame = new JFrame();
8        frame.setSize(300, 400);
9        frame.setTitle("Two rectangles");
10       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11
12       RectangleComponent component = new RectangleComponent();
13       frame.add(component);
14
```

```
15        frame.setVisible(true);
16     }
17  }
```

Observe que o programa de desenho de retângulo consiste em duas classes:

- A classe `RectangleComponent`, cujo método `paintComponent` produz o desenho.
- A classe `RectangleViewer`, cujo método `main` constrói um frame e um `RectangleComponent`, adiciona o componente ao frame e o torna visível.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

29. Como você modificaria o programa para desenhar dois quadrados?

30. Como você modificaria o programa para desenhar um retângulo e um quadrado?

31. O que acontece se você chamar `g.draw(box)` em vez de `g2.draw(box)`?

TÓPICO AVANÇADO 2.2

Applets

O Tópico Avançado 2.2 mostra como implementar programas que exibem desenhos como applets, ou seja, programas que executam dentro de um navegador Web.

2.13 Elipses, linhas, texto e cores

Na Seção 2.12, você aprendeu a escrever um programa que desenha retângulos. Nesta seção, você aprenderá a desenhar outras formas: elipses e linhas. Com esses elementos gráficos, você pode desenhar muitas figuras interessantes.

Para desenhar uma elipse, você especifica a caixa delimitadora (veja Figura 23) da mesma maneira como você especificaria um retângulo, a saber, pelas coordenadas *x* e *y* do canto superior esquerdo e pela largura e altura da caixa.

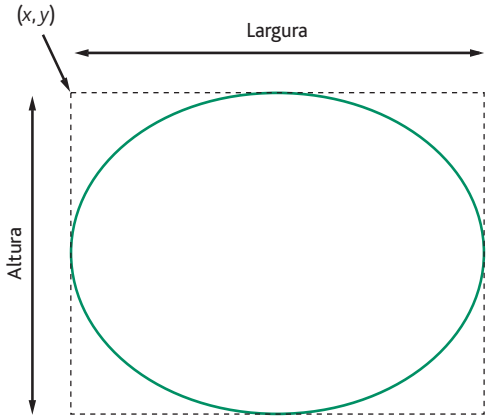


Figura 23 Uma elipse e sua caixa delimitadora.

Entretanto, nenhuma classe `Ellipse` simples está disponível para que você possa utilizar. Em vez disso, você deve utilizar uma das duas classes `Ellipse2D.Float` e `Ellipse2D.Double`, dependendo se você quer armazenar as coordenadas da elipse como valores de ponto flutuante de precisão simples ou dupla. Como os últimos são mais convenientes de utilizar em Java, sempre utilizaremos a classe `Ellipse2D.Double`. Eis como você constrói uma elipse:

```
Ellipse2D.Double ellipse = new Ellipse2D.Double(x, y, width, height);
```

`Ellipse2D.Double` e `Line2D.Double` são classes que descrevem formas gráficas.

O nome da classe `Ellipse2D.Double` parece diferente dos nomes de classe que vimos até agora. Ele consiste em dois nomes de classe `Ellipse2D` e `Double` separados por um ponto (`.`). Isso indica que `Ellipse2D.Double` é uma *classe interna,* dentro de `Ellipse2D`. Ao construir e utilizar elipses, na verdade você não precisa se preocupar com o fato de que `Ellipse2D.Double` é uma classe interna – simplesmente pense nela como uma classe com um nome longo. Mas, na instrução `import` na parte superior do seu programa, você deve ter cuidado em importar apenas a classe externa:

```
import java.awt.geom.Ellipse2D;
```

Desenhar uma elipse é fácil: utilize exatamente o mesmo método `draw` da classe `Graphics2D` que você utilizou para desenhar os retângulos.

```
g2.draw(ellipse);
```

Para desenhar um círculo, simplesmente configure a largura e a altura com os mesmos valores:

```
Ellipse2D.Double circle = new Ellipse2D.Double(x, y, diameter, diameter);
g2.draw(circle);
```

Observe que (`x`, `y`) é o canto superior esquerdo da caixa delimitadora, não o centro do círculo.

Para desenhar uma linha, utilize um objeto da classe `Line2D.Double`. Uma linha é construída especificando-se seus dois pontos finais. Você pode fazer isso de duas maneiras. Simplesmente forneça as coordenadas *x* e *y* dos dois pontos finais:

```
Line2D.Double segment = new Line2D.Double(x1, y1, x2, y2);
```

Ou especifique cada ponto final como um objeto da classe `Point2D.Double`:

```
Point2D.Double from = new Point2D.Double(x1, y1);
Point2D.Double to = new Point2D.Double(x2, y2);

Line2D.Double segment = new Line2D.Double(from, to);
```

A segunda opção é mais orientada a objetos e costuma ser mais útil, especialmente se os objetos ponto puderem ser reutilizados em outra parte no mesmo desenho.

Figura 24 Ponto de base e Linha de base.

O método `drawString` desenha uma string, iniciando no seu ponto de base.

Na maioria das vezes, queremos colocar texto dentro de um desenho, por exemplo, rotular alguns elementos. Utilize o método `drawString` da classe `Graphics2D` para desenhar uma string em qualquer lugar de uma janela. Você deve especificar a string e as coordenadas *x* e *y* do ponto de base do primeiro caractere na string (veja Figura 24). Por exemplo,

```
g2.drawString("Message", 50, 100);
```

2.13.1 Cores

Quando você começa a desenhar pela primeira vez, todas as formas e strings são desenhadas com uma caneta preta. Para alterar essa cor, você precisa fazer uso de um objeto do tipo `Color`. Java usa o modelo de cores RGB. Isto é, você especifica uma cor de acordo com os valores das cores primárias – vermelho, verde e azul – que compõem a cor. Os valores são fornecidos como inteiros entre 0 (ausência de cor primária) e 255 (valor máximo presente). Por exemplo,

```
Color magenta = new Color(255, 0, 255);
```

constrói um objeto `Color` com vermelho máximo, nenhum verde e azul máximo, produzindo uma cor roxa brilhante chamada magenta.

Quando você configura uma nova cor no contexto gráfico, ela é utilizada para as operações de desenho subseqüentes.

Para sua conveniência, uma variedade de cores foi predefinida na classe `Color`. A Tabela 1 mostra essas cores predefinidas e seus valores RGB. Por exemplo, `Color.PINK` foi predefinida como sendo a mesma cor de `new Color(255, 175, 175)`.

Para desenhar um retângulo com uma cor diferente, primeiro configure a cor do objeto `Graphics2D` e então chame o método `draw`:

```
g2.setColor(Color.RED);
g2.draw(circle); // desenha a forma em vermelho
```

Tabela 1 Cores predefinidas e seus valores RGB

Cor	Valor RGB
`Color.BLACK`	0, 0, 0
`Color.BLUE`	0, 0, 255
`Color.CYAN`	0, 255, 255
`Color.GRAY`	128, 128, 128
`Color.DARKGRAY`	64, 64, 64
`Color.LIGHTGRAY`	192, 192, 192
`Color.GREEN`	0, 255, 0
`Color.MAGENTA`	255, 0, 255
`Color.ORANGE`	255, 200, 0
`Color.PINK`	255, 175, 175
`Color.RED`	255, 0, 0
`Color.WHITE`	255, 255, 255
`Color.YELLOW`	255, 255, 0

Se quiser colorir o interior da forma, utilize o método `fill` em vez do método `draw`. Por exemplo,

```
g2.fill(circle);
```

preenche o interior do círculo com a cor atual.

O programa a seguir coloca todas essas formas em funcionamento, criando um desenho simples (veja Figura 25).

ch02/faceviewer/FaceComponent.java

```
1  import java.awt.Color;
2  import java.awt.Graphics;
3  import java.awt.Graphics2D;
4  import java.awt.Rectangle;
5  import java.awt.geom.Ellipse2D;
6  import java.awt.geom.Line2D;
7  import javax.swing.JPanel;
8  import javax.swing.JComponent;
9
10 /**
11    Um componente que desenha o rosto de um alienígena.
12 */
13 public class FaceComponent extends JComponent
14 {
15    public void paintComponent(Graphics g)
16    {
17       // Recupera Graphics2D
18       Graphics2D g2 = (Graphics2D) g;
19
20       // Desenha a cabeça
21       Ellipse2D.Double head = new Ellipse2D.Double(5, 10, 100, 150);
22       g2.draw(head);
23
24       // Desenha os olhos
25       Line2D.Double eye1 = new Line2D.Double(25, 70, 45, 90);
26       g2.draw(eye1);
27
28       Line2D.Double eye2 = new Line2D.Double(85, 70, 65, 90);
29       g2.draw(eye2);
30
31       // Desenha a boca
32       Rectangle mouth = new Rectangle(30, 130, 50, 5);
33       g2.setColor(Color.RED);
34       g2.fill(mouth);
35
36       // Desenha a saudação
37       g2.setColor(Color.BLUE);
38       g2.drawString("Hello, World!", 5, 175);
39    }
40 }
```

ch02/faceviewer/FaceViewer.java

```
1  import javax.swing.JFrame;
2
3  public class FaceViewer
4  {
5     public static void main(String[] args)
6     {
7        JFrame frame = new JFrame();
8        frame.setSize(300, 400);
9        frame.setTitle("An Alien Face");
10       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
11
12       FaceComponent component = new FaceComponent();
13       frame.add(component);
14
15       frame.setVisible(true);
16    }
17 }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

32. Forneça instruções para desenhar um círculo com centro (100, 100) e raio 25.
33. Forneça instruções para desenhar uma letra "V" usando dois segmentos de linha.
34. Forneça instruções para desenhar uma string com a letra "V".
35. Quais são os valores das cores RGB de `Color.BLUE`?
36. Como você desenha um quadrado amarelo contra um fundo vermelho?

Figura 25 Rosto de um alienígena.

FATO ALEATÓRIO 2.2

A evolução da internet

O Fato Aleatório 2.2 investiga a evolução da internet desde seu humilde começo como uma rede de pesquisa até seu crescimento explosivo que iniciou quando Marc Andreesen, um estudante de pós-graduação, lançou o primeiro navegador Web.

RESUMO DO CAPÍTULO

1. Em Java, cada valor tem um tipo.
2. Você usa variáveis para armazenar valores que deseja utilizar em um momento posterior.
3. Identificadores para variáveis, métodos e classes são compostos de letras, dígitos e caracteres de sublinhado.
4. Por convenção, os nomes das variáveis devem iniciar com letra minúscula.
5. Utilize o operador de atribuição (`=`) para alterar o valor de uma variável.
6. Todas as variáveis devem ser inicializadas antes de você acessá-las.
7. Objetos são entidades no seu programa que você manipula chamando métodos.
8. Um método é uma seqüência de instruções que acessam os dados de um objeto.
9. Uma classe define os métodos que você pode aplicar aos objetos dela.
10. A interface pública de uma classe especifica o que você pode fazer com objetos dela. A implementação oculta descreve como essas ações são executadas.
11. Um parâmetro é uma entrada para um método.
12. O parâmetro implícito de uma chamada de método é o objeto em que o método é invocado.
13. O valor de retorno de um método é um resultado que o método calculou para uso pelo código que o chamou.
14. Um nome de método é sobrecarregado se uma classe tiver mais de um método com o mesmo nome (mas tipos diferentes de parâmetros).
15. O tipo `double` indica números de ponto flutuante que podem ter partes fracionárias.
16. Em Java, números não são objetos e tipos numéricos não são classes.
17. Os números podem ser combinados por operadores aritméticos como `+`, `-` e `*`.
18. Utilize o operador `new`, seguido por um nome de classe e parâmetros, para construir novos objetos.
19. Um método de acesso não altera o estado do seu parâmetro implícito. Um método modificador altera o estado.
20. Determinar o resultado esperado antecipadamente é uma parte importante do processo de teste.
21. Classes Java são agrupadas em pacotes. Utilize a instrução `import` para usar classes definidas em outros pacotes.
22. A documentação da API (*Application Programming Interface*) lista as classes e métodos da biblioteca Java.

23. Uma referência a objeto descreve a localização de um objeto.
24. Múltiplas variáveis de objeto podem conter referências ao mesmo objeto.
25. Variáveis numéricas armazenam números. Variáveis de objeto armazenam referências.
26. Para mostrar um frame, construa um objeto `JFrame`, configure seu tamanho e torne-o visível.
27. Para exibir um desenho em um frame, defina uma classe que estende a classe `JComponent`.
28. Coloque as instruções de desenho dentro do método `paintComponent`. Esse método é chamado sempre que o componente precisa ser redesenhado.
29. A classe `Graphics` permite manipular o estado dos elementos gráficos (como a cor atual).
30. A classe `Graphics2D` possui métodos para desenhar formas.
31. Utilize uma coerção (ou *typecasting*) para recuperar o objeto `Graphics2D` a partir do parâmetro `Graphics` do método `paintComponent`.
32. `Ellipse2D.Double` e `Line2D.Double` são classes que descrevem formas gráficas.
33. O método `drawString` desenha uma string, iniciando no seu ponto de base.
34. Quando você configura uma nova cor no contexto gráfico, ela é utilizada nas operações subseqüentes de desenho.

LEITURA ADICIONAL

1. `http://java.sun.com/javase/6/docs/api/index.html` A documentação da API Java.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.awt.Color
java.awt.Component
   getHeight
   getWidth
   setSize
   setVisible
java.awt.Frame
   setTitle
java.awt.geom.Ellipse2D.Double
java.awt.geom.Line2D.Double
java.awt.geom.Point2D.Double
java.awt.Graphics
   setColor
java.awt.Graphics2D
   draw
   drawString
   fill
java.awt.Rectangle
   translate
   getX
   getY
   getHeight
   getWidth
java.lang.String
   length
   replace
   toLowerCase
   toUpperCase
javax.swing.JComponent
   paintComponent
javax.swing.JFrame
   setDefaultCloseOperation
```

EXERCÍCIOS DE REVISÃO

★ **Exercício R2.1.** Explique a diferença entre um objeto e uma referência a objeto.

★ **Exercício R2.2.** Explique a diferença entre um objeto e uma variável de objeto.

★ **Exercício R2.3.** Explique a diferença entre um objeto e uma classe.

★★ **Exercício R2.4.** Forneça o código Java para construir um *objeto* da classe `Rectangle` e para declarar uma variável de objeto da classe `Rectangle`.

★★ **Exercício R2.5.** Explique a diferença entre o símbolo `=` em Java e na matemática.

★★★ **Exercício R2.6.** Variáveis não-inicializadas podem ser um problema sério. Você sempre deve inicializar cada variável `int` ou `double` com zero? Explique as vantagens e as desvantagens dessa estratégia.

★★ **Exercício R2.7.** Forneça o código Java para construir os seguintes objetos:

a. Um retângulo com centro (100, 100) e todos os comprimentos de lado iguais a 50.

b. Uma string `"Hello, Dave!"`.

Crie objetos, não variáveis de objeto.

★★ **Exercício R2.8.** Repita o Exercício R2.7, mas desta vez defina as variáveis de objeto que são inicializadas com os objetos requeridos.

★★ **Exercício R2.9.** Encontre os erros nestas instruções:

a.
```
Rectangle r = (5, 10, 15, 20);
```

b.
```
double width = Rectangle(5, 10, 15, 20).getWidth();
```

c.
```
Rectangle r;
r.translate(15, 25);
```

d.
```
r = new Rectangle();
r.translate("far, far away!");
```

★ **Exercício R2.10.** Identifique dois métodos de acesso e dois métodos modificadores da classe `Rectangle`.

★★ **Exercício R2.11.** Examine a documentação da API da classe `Rectangle` e localize o método.

```
void add(int newx, int newy)
```

Analise a documentação do método. Depois, determine o resultado das instruções a seguir:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
box.add(0, 0);
```

Se você não estiver seguro, escreva um pequeno programa de teste ou use o BlueJ.

★ **Exercício R2.12.** Encontre um método sobrecarregado da classe `String`.

★ **Exercício R2.13.** Localize um método sobrecarregado da classe `Rectangle`.

★G **Exercício R2.14.** Qual é a diferença entre uma aplicação de console e uma aplicação gráfica?

★★G **Exercício R2.15.** Quem chama o método `paintComponent` de um componente? Quando ocorre a chamada ao método `paintComponent`?

★★G **Exercício R2.16.** Por que o parâmetro do método `paintComponent` tem um tipo `Graphics` e não um `Graphics2D`?

★★G **Exercício R2.17.** Qual é o propósito de um contexto gráfico?

★★G **Exercício R2.18.** Por que são utilizadas uma classe de visualizadores e uma classe de componentes separadas para programas gráficos?

★G **Exercício R2.19.** Como você especifica uma cor de texto?

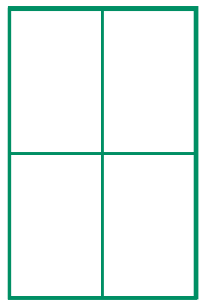
Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

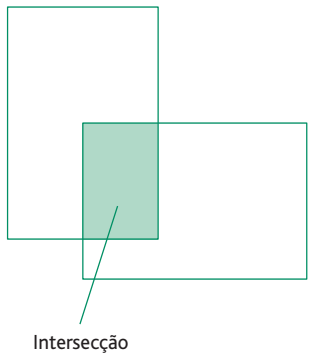
★T **Exercício P2.1.** Escreva um programa `AreaTester` que constrói um objeto `Rectangle` e então calcula e imprime sua área. Utilize os métodos `getWidth` e `getHeight`. Imprima a resposta esperada.

★T **Exercício P2.2.** Escreva um programa `PerimeterTester` que constrói um objeto `Rectangle` e então calcula e imprime seu perímetro. Utilize os métodos `getWidth` e `getHeight`. Imprima a resposta esperada.

★★ **Exercício P2.3.** Escreva um programa chamado `FourRectanglePrinter` que constrói um objeto `Rectangle`, imprime sua localização chamando `System.out.println(box)` e então o traduz e imprime mais três vezes, de modo que, se os retângulos fossem desenhados, eles formariam um grande retângulo:



★★★ **Exercício P2.4.** O método `intersection` calcula a *intersecção* de dois retângulos – isto é, o retângulo formado por dois retângulos que se sobrepõem:



Você chama esse método como a seguir:

```
Rectangle r3 = r1.intersection(r2);
```

Escreva um programa `IntersectionPrinter` que constrói dois objetos retângulo, os imprime e então imprime o objeto retângulo que descreve a interseção. Depois, o programa deve imprimir o resultado do método `intersection` quando os retângulos não se sobrepuserem. Adicione um comentário ao seu programa que explica como é possível dizer se o retângulo resultante está vazio.

★★ **Exercício P2.5.** Na biblioteca Java, uma cor é especificada pelos seus componentes de vermelho, verde e azul com valores entre 0 e 255. Escreva um programa `BrighterDemo` que constrói um objeto `Color` com valores de vermelho, verde e azul de 50, 100 e 150. Aplique então o método `brighter` e imprima os valores de vermelho, verde e azul da cor resultante. (Você não verá realmente a cor – consulte a Seção 2.13 sobre como exibir cores.)

★★ **Exercício P2.6.** Repita o Exercício P2.5, mas aplique o método `darker` duas vezes ao objeto predefinido `Color.RED`. Chame sua classe `DarkerDemo`.

★★ **Exercício P2.7.** A classe `Random` implementa um *gerador de números aleatórios*, que produz seqüências de números que parecem ser aleatórios. Para gerar inteiros aleatórios, você constrói um objeto da classe `Random` e então aplica o método `nextInt`. Por exemplo, a chamada `generator.nextInt(6)` fornece um número aleatório entre 0 e 5.

Escreva um programa `DieSimulator` que usa a classe `Random` para simular o lançamento de um dado, imprimindo um número aleatório entre 1 e 6 toda vez que o programa é executado.

★★★ **Exercício P2.8.** Escreva um programa `LotteryPrinter` que seleciona uma combinação em uma loteria. Nessa loteria, os apostadores podem escolher 6 números (possivelmente repetidos) entre 1 e 49. (Em uma loteria real, repetições não são permitidas, mas ainda não discutimos as construções de programação que seriam necessárias para lidar com esse problema.) Seu programa deve imprimir uma frase como "Jogue essa combinação – ela o tornará rico!", seguida por uma combinação de números.

★★T **Exercício P2.9.** Escreva um programa `ReplaceTester` que codifica uma string substituindo todas as letras `"i"` por `"!"` e todas as letras `"s"` por `"$"`. Utilize o método `replace`. Demonstre que você pode codificar corretamente a string `"Mississippi"`. Imprima o resultado real e o esperado.

★★★ **Exercício P2.10.** Escreva um programa `HollePrinter` que muda as letras `"e"` e `"o"` em uma string. Utilize o método `replace` repetidamente. Demonstre que a string `"Hello, World!"` transforma-se em `"Holle, Werld!"`.

★★G **Exercício P2.11.** Escreva um programa gráfico que desenhe seu nome em vermelho, contido dentro de um retângulo azul. Forneça uma classe `NameViewer` e uma classe `NameComponent`.

★★G **Exercício P2.12.** Escreva um programa gráfico que desenhe 12 strings, uma para cada uma das 12 cores padrão, além de `Color.WHITE`, e cada uma em sua própria cor. Forneça uma classe `ColorNameViewer` e uma classe `ColorNameComponent`.

★★G **Exercício P2.13.** Escreva um programa que desenha dois quadrados sólidos: um em cor-de-rosa e um em roxo. Utilize uma cor padrão para um deles e uma cor personalizada para o outro. Forneça uma classe `TwoSquareViewer` e uma classe `TwoSquareComponent`.

★★★G **Exercício P2.14.** Escreva um programa que preencha a janela com uma grande elipse, com um contorno preto e preenchida com sua cor favorita. A elipse deverá tocar os limites da janela, mesmo se a janela for redimensionada.

★★G **Exercício P2.15.** Escreva um programa para plotar o rosto a seguir.

Forneça uma classe `FaceViewer` e uma classe `FaceComponent`.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 2.1.** A classe `GregorianCalendar` descreve um ponto no tempo, como medido pelo calendário gregoriano, o calendário padrão comumente utilizado em quase todo o mundo atual. Você constrói um objeto `GregorianCalendar` a partir de um ano, mês e dia, como no exemplo a seguir:

```
GregorianCalendar cal = new GregorianCalendar(); // Data atual
GregorianCalendar eckertsBirthday = new GregorianCalendar(1919,
      Calendar.APRIL, 9);
```

Utilize as constantes `Calendar.JANUARY . . . Calendar.DECEMBER` para especificar o mês.

O método `add` pode ser utilizado para adicionar alguns dias a um objeto `GregorianCalendar`:

```
cal.add(Calendar.DAY_OF_MONTH, 10); // Agora cal é dez dias depois de hoje
```

Esse é um método modificador – ele altera o objeto `cal`.

O método `get` pode ser utilizado para consultar um objeto `GregorianCalendar`:

```
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);
int month = cal.get(Calendar.MONTH);
int year = cal.get(Calendar.YEAR);
int weekday = cal.get(Calendar.DAY_OF_WEEK);
   // 1 é domingo, 2 é segunda-feira,. . . , 7 é sábado
```

Sua tarefa é escrever um programa que imprime estas informações:

- A data e o dia da semana daqui a 100 dias a partir da data atual
- O dia da semana do seu aniversário
- A data daqui a 10.000 dias a partir do seu aniversário

Utilize o aniversário de um cientista da computação se você não quiser revelar seu próprio aniversário.

★★★G **Projeto 2.2.** Execute o programa a seguir:

```
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JTextField;

public class FrameTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      frame.setSize(200, 200);
      JTextField text = new JTextField("Hello, World!");
      text.setBackground(Color.PINK);
      frame.add(text);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

Modifique o programa desta maneira:

- Dobre o tamanho do frame
- Mude a saudação para "Hello, *seu nome*!"
- Mude a cor de fundo para verde pálido (veja o Exercício P2.5)

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. `int` e `String`
2. Somente os dois primeiros são identificadores válidos.
3. `String myName = "John Q. Public";`
4. Não, o lado esquerdo do operador `=` deve ser uma variável.
5. `greeting = "Hello, Nina!";`

   Observe que:

   `String greeting = "Hello, Nina!";`

   não é a resposta correta – essa instrução define uma nova variável.
6. `river.length()` ou `"Mississippi".length()`
7. `System.out.println(greeting.toUpperCase());`
8. Não é válido. A variável `river` é do tipo `String`. O método `println` não é um método da classe `String`.
9. O parâmetro implícito é `river`. Não há parâmetro explícito. O valor de retorno é 11.
10. `"Missississi"`
11. 12
12. Como `public String toUpperCase()`, sem parâmetro explícito e sem tipo de retorno `String`.
13. `double`
14. Um `int` não é um objeto e você não pode chamar um método associado a ele.

15. `(x + y) * 0.5`
16. `new Rectangle(90, 90, 20, 20)`
17. 0
18. Um método de acesso – não modifica a string original, mas retorna uma nova string em letras maiúsculas.
19. `box.translate(-5, -10)`, desde que o método seja chamado logo depois de armazenar o novo retângulo em `box`.
20. `x: 30, y: 25`
21. Porque o método `translate` não modifica a forma do retângulo.
22. Adicione a instrução `import java.util.Random;` na parte superior de seu programa.
23. `toLowerCase`
24. `"Hello, Space !"` – somente os espaços iniciais e finais são cortados.
25. Agora `greeting` e `greeting2` referenciam o mesmo objeto `String`.
26. As duas variáveis ainda referem-se à mesma string e a string não foi modificada. Lembre-se de que o método `toUpperCase` constrói uma nova string que contém caracteres em letras maiúsculas, deixando a string original inalterada.
27. Modifique o programa `EmptyFrameViewer` como mostrado a seguir:

```
frame.setSize(300, 300);
frame.setTitle("Hello, World!");
```

28. Construa dois objetos `JFrame`, configure cada um dos seus tamanhos e chame `setVisible(true)` em cada um deles.
29. `Rectangle box = new Rectangle(5, 10, 20, 20);`
30. Substitua a chamada a `box.translate(15, 25)` por

```
box = new Rectangle(20, 35, 20, 20);
```

31. O compilador reclama que `g` não tem um método `draw`.
32. `g2.draw(new Ellipse2D.Double(75, 75, 50, 50));`
33. `Line2D.Double segment1 = new Line2D.Double(0, 0, 10, 30);`

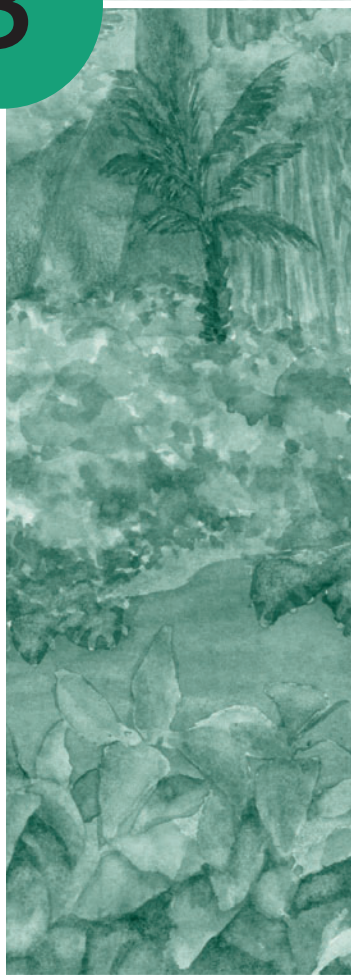
```
g2.draw(segment1);
Line2D.Double segment2 = new Line2D.Double(10, 30, 20, 0);
g2.draw(segment2);
```

34. `g2.drawString("V", 0, 30);`
35. `0, 0, 255`
36. Primeiro preencha um grande quadrado vermelho e então preencha um pequeno quadrado amarelo dentro:

```
g2.setColor(Color.RED);
g2.fill(new Rectangle(0, 0, 200, 200));
g2.setColor(Color.YELLOW);
g2.fill(new Rectangle(50, 50, 100, 100));
```

Capítulo 3

Implementando Classes



Objetivos do capítulo

- Familiarizar-se com o processo de implementação de classes
- Ser capaz de implementar métodos simples
- Entender o propósito e o uso de construtores
- Entender como acessar campos de instância e variáveis locais
- Entender a importância dos comentários de documentação
- G Implementar classes para desenhar formas gráficas

Neste capítulo, você aprenderá a implementar suas próprias classes. Você começará com um projeto dado que especifica a interface pública da classe – isto é, os métodos por meio dos quais programadores podem manipular os objetos da classe. Então, você precisará implementar os métodos. Esse passo requer que você encontre uma representação de dados para os objetos e forneça as instruções para cada método. Em seguida, você fornecerá um testador para comprovar que sua classe funciona corretamente. Também documentará seus esforços para que outros programadores possam entender e utilizar sua criação.

Conteúdo do Capítulo

3.1 Níveis de abstração 100

3.2 Especificando a interface pública de uma classe 103

Sintaxe 3.1: Definição de método 105
Sintaxe 3.2: Definição de construtor 106
Sintaxe 3.3: Definição de classe 107

3.3 Comentando a interface pública 107

Dica de produtividade 3.1: O utilitário `javadoc` ⊕

3.4 Campos de instância 110

Sintaxe 3.4: Declaração de campo de instância 112

3.5 Implementando construtores e métodos 112

Sintaxe 3.5: A instrução `return` 114
Como fazer 3.1: Implementando uma classe 115

3.6T Teste de unidade 119

Dica de produtividade 3.2: Utilizando a linha de comando de maneira eficiente ⊕

3.7 Categorias de variáveis 121

Erro comum 3.1: Esquecendo de inicializar referências a objetos em um construtor 123

3.8 Parâmetros de método implícitos e explícitos 124

Erro comum 3.2: Tentando chamar um método sem um parâmetro implícito 125
Tópico avançado 3.1: Chamando um construtor a partir de um outro ⊕
Fato aleatório 3.1: Urnas eletrônicas ⊕

3.9G Classes de formas gráficas 126

Como fazer 3.2: Desenhando formas gráficas 131
Fato aleatório 3.2: Computação gráfica ⊕

3.1.1 Caixas pretas

Ao levantar o capô de um carro, você encontrará uma coleção enorme de componentes mecânicos. Você provavelmente reconhecerá o motor e o recipiente de água do limpador de pára-brisa. Um mecânico será capaz de identificar vários outros componentes, como a transmissão e o módulo de controle eletrônico – o dispositivo que controla a sincronização das velas de ignição e o fluxo de gasolina no motor. Mas pergunte ao seu mecânico o que está dentro do módulo de controle eletrônico e provavelmente ele não saberá o que responder.

É uma *caixa preta*, algo que funciona de uma maneira mágica. Um mecânico nunca abriria a caixa – ela contém componentes eletrônicos que só podem ser revisados na fábrica. Naturalmente, o dispositivo pode ter uma outra cor além do preto e talvez nem mesmo tenha uma forma de caixa. Mas os engenheiros utilizam o termo "caixa preta" para descrever qualquer dispositivo cujo funcionamento interno permanece oculto. Observe que uma caixa preta não é totalmente misteriosa. Sua interação com o mundo externo é bem-definida. Por exemplo, o mecânico pode testar se o módulo de controle do motor envia a descarga eletrônica correta para as velas de ignição.

Por que os fabricantes colocam caixas pretas nos carros? A caixa preta simplifica significativamente o funcionamento da mecânica do carro, o que resulta em custos de manutenção mais baixos. Se a caixa falhar, é simplesmente substituída por uma nova. Antes dos módulos de controle do motor terem sido inventados, o fluxo de gasolina no motor era regulado por um dispositivo mecânico chamado carburador, um grande caos de molas e engates caros de ajustar e consertar.

Naturalmente, para muitos motoristas, o *carro inteiro* é uma "caixa preta". A maioria deles não conhece nada sobre o funcionamento interno e nunca vai querer abrir o capô. O carro tem pedais, botões e uma tampa de tanque de gasolina. Se você fornecer as entradas corretas, ele fará o que deve ser feito, transportá-lo de um lugar a outro.

E para o fabricante do módulo de controle do motor, os transistores e capacitores internos são caixas pretas, magicamente produzidas pelo fabricante do componente eletrônico.

Em termos técnicos, uma caixa preta fornece *o encapsulamento*, escondendo os detalhes sem importância. O encapsulamento é muito importante na resolução de problemas por humanos. Um mecânico é mais eficiente quando a única decisão é testar o módulo de controle eletrônico e substituí-lo quando ele falha, sem que precise pensar nos sensores e transistores internos. Um motorista é mais eficiente quando a única preocupação é encher o tanque, não pensar no motor ou no módulo de controle eletrônico interno.

Mas há um outro aspecto do encapsulamento. Alguém tem de propor o *conceito* correto para cada caixa preta. Por que os fabricantes de peças automotivas constroem módulos de controle eletrônico e não um outro dispositivo? Por que os fabricantes de carros constroem carros e não helicópteros pessoais?

Os conceitos são descobertos por meio do processo de *abstração*, excluindo os recursos não-essenciais até que apenas a essência do conceito permaneça. Por exemplo, "carro" é uma abstração, descrevendo dispositivos que transportam pequenos grupos de pessoas que viajam por via terrestre e consomem gasolina. Essa é a abstração correta? Ou um veículo com um motor elétrico seria um "carro"? Não responderemos a essa pergunta e, em vez disso, analisaremos a importância do encapsulamento e da abstração na ciência da computação.

3.1.2 Projeto orientado a objetos

Antigamente, programas de computador manipulavam *tipos primitivos* como números e caracteres. À medida que esses programas tornaram-se mais complexos, passaram a manipular um número cada vez maior desses tipos primitivos até o ponto em que os programadores não conseguiam mais acompanhar esse processo. Era demasiadamente confuso memorizar todos esses detalhes. Como resultado, programadores davam instruções erradas aos computadores e estes as executavam rigorosamente conforme instruídos, produzindo respostas também erradas.

Naturalmente, a resposta a esse problema era óbvia. Os desenvolvedores de software aprenderam rapidamente a gerenciar a complexidade. Eles encapsularam cálculos rotineiros, criando "caixas pretas" de software que poderiam ser colocadas em funcionamento sem a preocupação com as partes internas. Eles utilizavam o processo de abstração para criar tipos de dados que estão em um nível mais alto do que números e caracteres.

No momento em que este livro estava sendo escrito, a abordagem mais comum para estruturar a programação de computadores era a abordagem *orientada a objetos*. As caixas pretas a partir das quais um programa é produzido são chamadas objetos. Um objeto tem uma estrutura interna – talvez apenas alguns números, talvez outros objetos – e um comportamento bem-definido. Naturalmente, a estrutura interna permanece escondida do programador que a usa. Esse programador só aprende o comportamento do objeto e então o coloca em funcionamento para alcançar um objetivo de nível mais alto.

Quem projeta esses objetos? Outros programadores! O que eles contêm? Outros objetos! É aqui que as coisas começam a ficar confusas para estudantes iniciantes. Na vida real, os usuários das caixas pretas são bem diferentes dos projetistas e é fácil entender os níveis de abstração (veja Figura 1). Com programas de computador, também há níveis de abstração (veja Figura 2), mas eles não são tão intuitivos aos não-iniciados. Para tornar as coisas ainda mais confusas, freqüentemente você precisará trocar de papel, sendo o projetista de objetos pela manhã e o usuário desses mesmos objetos à tarde. Nesse sentido, você será como os primeiros fabricantes de automóveis, que produziam, sem ajuda, volantes e eixos e então incorporavam suas criações a um carro.

Há outro aspecto desafiador no projeto de objetos. O software é infinitamente mais flexível do que o hardware porque o software está livre das limitações físicas. Projetistas de componentes eletrônicos podem explorar um número limitado de efeitos físicos para criar transistores, capacitores e assim por diante. Fabricantes de automóveis não conseguem produzir facilmente helicópteros pessoais devido a uma infinidade de limitações físicas, por exemplo, consumo de combustível e segurança. Mas, em software, qualquer coisa é possível. Com poucas restrições externas, você pode projetar abstrações boas e ruins com igual facilidade. Entender o que constitui um bom projeto é uma parte importante da educação do engenheiro de software.

3.1.3 Engatinhar, caminhar, correr

No Capítulo 2, você aprendeu a ser um usuário de objetos. Você viu como obter objetos, como manipulá-los e como usá-los em um programa. Naquele capítulo, seu papel era análogo ao do engenheiro automotivo que aprende a utilizar um módulo de controle de motor e a tirar proveito desse comportamento para construir um carro.

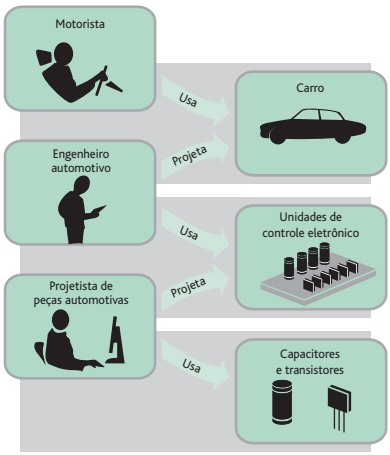


Figura 1 Níveis de abstração no projeto automotivo.

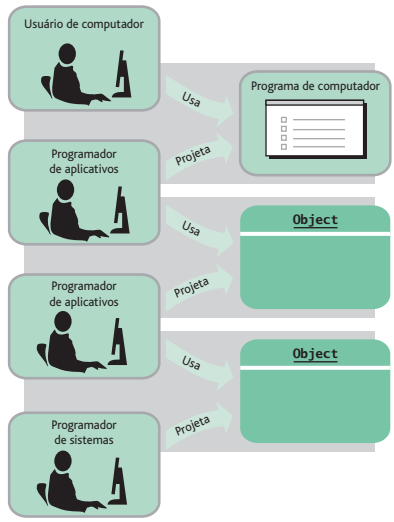


Figura 2 Níveis de abstração no projeto de software.

Neste capítulo, você passará para a implementação de classes. Forneceremos um projeto que descreve o comportamento dos objetos de uma classe. Você aprenderá as técnicas de programação necessárias em Java que permitem que seus objetos realizem o comportamento desejado. Nestas seções, seu papel será análogo ao do fabricante de peças automotivas que monta um módulo de controle do motor a partir dos transistores, capacitores e outros componentes eletrônicos.

Nos Capítulos 8 e 12, você aprenderá mais sobre como projetar suas próprias classes. Aprenderá também as regras de um bom projeto e como descobrir o comportamento apropriado dos objetos. Nesses capítulos, seu trabalho será análogo ao do engenheiro de peças automotivas que especifica como um módulo de controle do motor deve funcionar.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Nos Capítulos 1 e 2, você utilizou `System.out` como uma caixa preta para fazer a saída aparecer na tela. Quem projetou e implementou `System.out`?
2. Suponha que você esteja trabalhando em uma empresa que produz um software de finanças pessoais. Você foi encarregado de projetar e implementar uma classe para representar contas bancárias. Quem serão os usuários dessa classe?

3.2 Especificando a interface pública de uma classe

Nesta seção, discutiremos o processo de especificação do comportamento de uma classe. Imagine que você seja membro de uma equipe que trabalha com software de operações bancárias. Um conceito fundamental sobre operações bancárias é uma *conta bancária*. Sua tarefa é entender o projeto de uma classe `BankAccount` para que possa implementá-la, o que, por sua vez, permitirá que outros programadores na equipe a utilizem.

Para implementar uma classe, primeiro é necessário saber quais métodos são necessários.

Você precisa saber exatamente quais recursos de uma conta bancária precisam ser implementados. Alguns recursos são essenciais (por exemplo, depósitos), enquanto outros não são importantes (como o brinde que um cliente pode receber por abrir uma conta bancária). Decidir quais recursos são essenciais nem sempre é uma tarefa fácil. Vamos rever essa questão nos Capítulos 8 e 12. Por enquanto, iremos supor que um projetista competente decidiu que as operações a seguir são consideradas essenciais para uma conta bancária:

- Depositar dinheiro
- Sacar dinheiro
- Obter o saldo atual

Em Java, operações são expressas como chamadas de método. Para descobrir a especificação exata das chamadas de método, imagine como um programador executaria as operações da conta bancária. Iremos supor que a variável `harrysChecking` contém uma referência a um objeto do tipo `BankAccount`. Queremos suportar chamadas de método como as listadas abaixo:

```
harrysChecking.deposit(2000);
harrysChecking.withdraw(500);
System.out.println(harrysChecking.getBalance());
```

Observe que os dois primeiros métodos são modificadores. Eles alteram o saldo da conta bancária e não retornam um valor. O terceiro método é um método de acesso. Ele retorna um valor que pode ser impresso ou armazenado em uma variável.

Como você pode ver a partir das chamadas do exemplo, a classe `BankAccount` deve definir três métodos:

- `public void deposit(double amount)`
- `public void withdraw(double amount)`
- `public double getBalance()`

Lembre-se do Capítulo 2 que `double` significa um tipo de ponto flutuante de dupla precisão e `void` indica que um método não retorna valor algum.

Ao definir um método, você também precisa fornecer o *corpo* do método, que consiste em instruções que são executadas quando o método é chamado.

```
public void deposit(double amount)
{
   corpo – preenchido mais adiante
}
```

Veremos na Seção 3.5 como preencher o corpo do método. Cada definição de método contém as seguintes partes:

- Um *especificador de acesso* (normalmente `public`)
- O *tipo de retorno* (como `void` ou `double`)
- O nome do método (como `deposit`)
- Uma lista de *parâmetros* do método (se houver algum) entre parênteses (como `double amount`)
- O *corpo* do método: instruções entre chaves

Uma definição de método contém um especificador de acesso (normalmente `public`), um tipo de retorno, o nome do método, parâmetros e o corpo do método.

O especificador de acesso controla quais outros métodos podem invocar esse método. A maioria dos métodos deve ser declarada como `public`. Dessa maneira, todos os outros métodos em um programa podem chamá-los. (Ocasionalmente, pode ser útil ter métodos `private`. Eles só podem ser chamados a partir de outros métodos da mesma classe.)

O tipo de retorno é o tipo do valor de saída. O método `deposit` não retorna um valor e o método `getBalance` retorna um valor do tipo `double`.

Cada parâmetro (ou entrada) para o método tem um tipo e um nome. Por exemplo, o método `deposit` tem um único parâmetro chamado `amount` do tipo `double`. Para cada parâmetro, escolha um nome que seja tanto um nome de variável válido como uma boa descrição do propósito da entrada.

Em seguida, você precisa fornecer construtores. Criaremos contas bancárias com saldo inicial zero, utilizando o construtor padrão:

```
BankAccount harrysChecking = new BankAccount();
```

E se um programador que usa nossa classe quiser iniciar com um outro saldo? Um segundo construtor que configura o saldo como um valor inicial será útil:

```
BankAccount momsSavings = new BankAccount(5000);
```

Resumindo, foi especificado que serão fornecidos dois construtores:

SINTAXE 3.1 Definição de método

especificadorDeAcesso tipoDeRetorno nomeDoMétodo(tipoDeParâmetro nomeDoParâmetro, . . .)
```
{
    corpo do método
}
```

Exemplo:

```
public void deposit(double amount)
{
   . . .
}
```

Objetivo:

Definir o comportamento de um método

- `public BankAccount()`
- `public BankAccount(double initialBalance)`

Um construtor é muito semelhante a um método, com duas importantes diferenças.

- O nome do construtor sempre é igual ao nome da classe (por exemplo, `BankAccount`)
- Construtores não têm nenhum tipo de retorno (nem mesmo `void`)

Construtores contêm instruções para inicializar objetos. O nome do construtor sempre é igual ao nome da classe.

Assim como um método, um construtor também tem um corpo – uma seqüência de instruções que é executada quando um novo objeto é criado.

```
public BankAccount()
{
    corpo – preenchido mais adiante
}
```

As instruções no corpo do construtor configuram os dados internos do objeto que está sendo criado – veja a Seção 3.5.

Não se preocupe com o fato de haver dois construtores com o mesmo nome – *todos* os construtores de uma classe têm o mesmo nome, isto é, o nome da classe. O compilador pode diferenciá-los porque eles recebem parâmetros diferentes.

Ao definir uma classe, você coloca todas as definições do construtor e dos métodos dentro dela, da seguinte maneira:

```
public class BankAccount
{
   // Construtores
   public BankAccount()
   {
      corpo – preenchido mais adiante
   }
   public BankAccount(double initialBalance)
```

```
    {
        corpo – preenchido mais adiante
    }
    // Métodos
    public void deposit(double amount)
    {
        corpo – preenchido mais adiante
    }
    public void withdraw(double amount)
    {
        corpo – preenchido mais adiante
    }
    public double getBalance()
    {
        corpo – preenchido mais adiante
    }
    campos privados – preenchidos mais adiante
}
```

Veremos como fornecer as partes que faltam nas seções a seguir.

Os construtores e os métodos públicos de uma classe formam a *interface pública* dessa classe. Estas são as operações que qualquer programador pode utilizar para criar e manipular objetos `BankAccount`. Nossa classe `BankAccount` é simples, mas permite que programadores executem todas as operações importantes que comumente ocorrem com contas bancárias. Por exemplo, considere este segmento de programa, escrito por um programador que usa a classe `BankAccount`. Estas instruções transferem uma quantidade de dinheiro de uma conta bancária para outra:

```
// Transfere de uma conta para outra
double transferAmount = 500;
momsSavings.withdraw(transferAmount);
harrysChecking.deposit(transferAmount);
```

SINTAXE 3.2 Definição de construtor

especificadorDeAcesso NomeDaClasse(*tipoDeParâmetro nomeDoParâmetro*, . . .)
{
    corpo do construtor
}

Exemplo:

```
public BankAccount(double initialBalance)
{
   . . .
}
```

Objetivo:

Definir o comportamento de um construtor

SINTAXE 3.3 Definição de classe

especificadorDeAcesso `class` *NomeDaClasse*
{
    construtores
    métodos
    campos
}

Exemplo:

```
public class BankAccount
{
   public BankAccount(double initialBalance) { . . . }
   public void deposit(double amount) { . . . }
   . . .
}
```

Objetivo:

Definir uma classe, sua interface pública e os detalhes da sua implementação

E eis um segmento de um programa que adiciona juros a uma conta de poupança:

```
double interestRate = 5; // 5% de juros
double interestAmount
      = momsSavings.getBalance() * interestRate / 100;
momsSavings.deposit(interestAmount);
```

Como você pode ver, programadores podem utilizar objetos da classe `BankAccount` para executar tarefas significativas, sem saber como os objetos `BankAccount` armazenam dados ou como o método `BankAccount` funciona.

Naturalmente, como implementadores da classe `BankAccount`, precisaremos fornecer os detalhes internos. Faremos isso na Seção 3.5. Primeiro, porém, há um outro passo importante: *documentar* a interface pública. Esse é o tópico da próxima seção.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Como você pode utilizar os métodos da interface pública para *zerar* a conta bancária `harrysChecking`?
4. Suponha que você queira uma abstração mais poderosa da conta bancária que, além do saldo, monitore um *número de conta*. Como você alteraria a interface pública para acomodar esse aprimoramento?

3.3 Comentando a interface pública

Ao implementar classes e métodos, você deve ter o hábito de *comentar* cuidadosamente seus comportamentos. Em Java há uma forma padrão bem útil para os *comentários de documentação*. Se utilizar essa forma nas suas classes, um programa chamado `javadoc` poderá gerar automaticamente um conjunto elegante de páginas HTML

que descrevem esses comentários. (Veja a Dica de produtividade 3.1 para uma descrição desse utilitário.)

Utilize os comentários da documentação para descrever as classes e os métodos públicos dos seus programas.

Um comentário de documentação é inserido antes da definição da classe ou do método que está sendo documentado. Ele inicia com um `/**`, um delimitador especial de comentário utilizado pelo utilitário `javadoc`. Depois você descreve o *propósito* do método. Em seguida, para cada parâmetro de método, você fornece uma linha que inicia com `@param`, seguida pelo nome do parâmetro e por uma breve explicação. Por fim, você fornece uma linha que inicia com `@return`, descrevendo o valor de retorno. Você deve omitir a marca `@param` para métodos que não têm parâmetro, e a marca tag `@return` para métodos cujo tipo de retorno é `void`.

O utilitário `javadoc` copia a *primeira* frase de cada comentário para uma tabela de resumo na documentação em HTML. Portanto, é melhor escrever essa primeira frase com algum cuidado. Você deve iniciar com uma letra maiúscula e terminar com um ponto. Ela não tem de ser uma frase gramaticalmente completa, mas deve ser significativa quando for extraída do comentário e exibida em um resumo.

Eis dois exemplos típicos.

```
/**
   Retira dinheiro da conta bancária.
   @param amount valor a retirar
*/
public void withdraw(double amount)
{
   implementação – preenchida mais adiante
}
/**
   Obtém o saldo atual da conta bancária.
   @return saldo atual
*/
public double getBalance()
{
   implementação – preenchida mais adiante
}
```

Os comentários que você acaba de ver explicam *métodos* individuais. Forneça um breve comentário para cada *classe*, explicando o propósito. A sintaxe do comentário para comentários de classe é muito simples: apenas insira o comentário de documentação acima da classe.

```
/**
   Uma conta bancária tem um saldo que pode ser alterado através de
   depósitos e saques.
*/
public class BankAccount
{
   . . .
}
```

Sua primeira reação poderia muito bem ser "Uau! Preciso escrever tudo isso?" Esses comentários parecem bem repetitivos. Mas você deve parar e escrevê-los, mesmo se eles parecerem ridículos.

É sempre uma boa idéia escrever *primeiro* o comentário do método, antes de escrever o código no seu corpo. Esse é um excelente teste para ver se você entende perfeitamente o que precisa programar. Se não puder explicar o que uma classe ou método faz, você não está pronto para implementá-lo.

Forneça comentários de documentação para cada classe, cada método, cada parâmetro e cada valor de retorno.

E os métodos muito simples? Facilmente você pode gastar mais tempo ponderando se um comentário é muito trivial para ser escrito do que escrevendo-o. Na programação prática, métodos muito simples são raros. Não há problema se um método trivial for excessivamente comentado, enquanto que um método complexo sem nenhum comentário pode causar complicações reais para a manutenção futura pelos programadores. De acordo com o estilo Java de documentação padrão, *cada* classe, *cada* método, *cada* parâmetro e *cada* valor de retorno devem ter um comentário.

O utilitário `javadoc` formata seus comentários em um elegante conjunto de documentos que pode ser visualizado em um navegador Web. Ele faz bom uso de frases aparentemente repetitivas. A primeira frase do comentário é utilizada por uma *tabela de resumo* de todos os métodos da sua classe (veja Figura 3). Os comentários `@param` e `@return` são elegantemente formatados em uma descrição detalhada de cada método (veja Figura 4). Se omitir um dos comentários, então o `javadoc` gera documentos que parecem estranhamente vazios.

Esse formato de documentação deve parecer familiar. Os próprios programadores que implementam a biblioteca Java utilizam o `javadoc`. Eles também documentam todas as classes, métodos, parâmetros e valores de retorno e depois utilizam o `javadoc` para extrair a documentação no formato HTML.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

5. Suponha que a classe `BankAccount` seja aprimorada para que cada conta tenha um número. Forneça um comentário de documentação para o construtor.

```
public BankAccount(int accountNumber, double initialBalance)
```

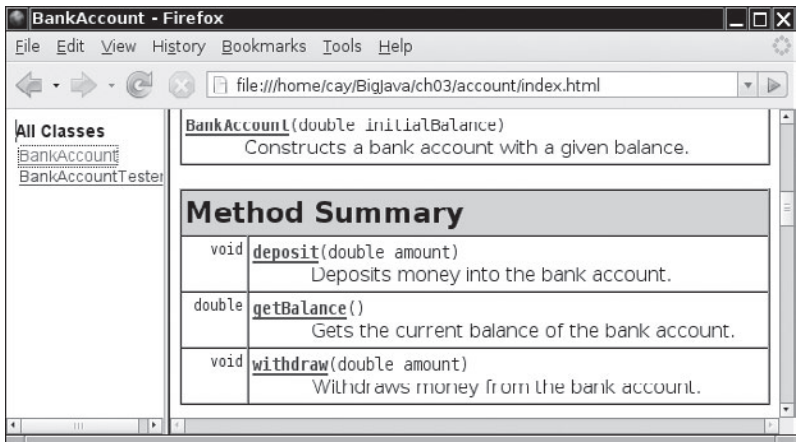


Figura 3 Um resumo de métodos gerado pelo `javadoc`.

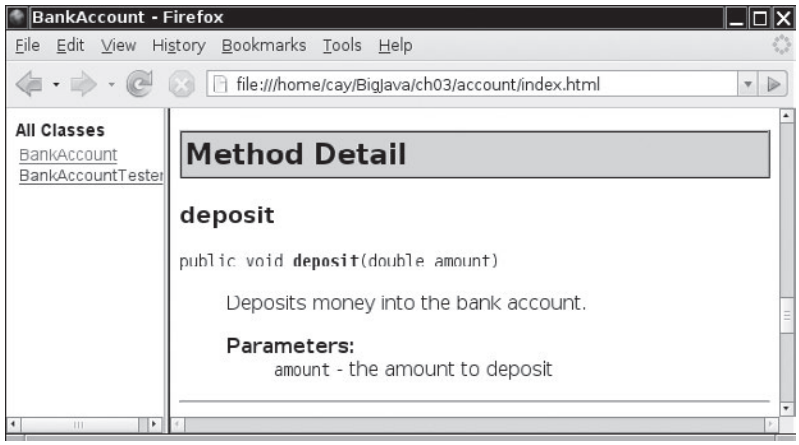


Figura 4 Detalhe de um método gerado pelo `javadoc`.

6. Por que o comentário de documentação a seguir é questionável?

```
/**
   Cada conta tem um número de conta.
   @return número de conta dessa conta
*/
public int getAccountNumber()
```

Dica de produtividade 3.1

O utilitário `javadoc`

A Dica de Produtividade 3.1 descreve como executar o utilitário `javadoc` que transforma os comentários de documentação em um conjunto de arquivos HTML.

3.4 Campos de instância

Agora que você entende a especificação da interface pública da classe `BankAccount`, vamos fornecer a implementação.

Primeiro, precisamos determinar os dados que cada objeto conta bancária contém. No caso da nossa classe de conta bancária simples, cada objeto precisa armazenar um único valor, o saldo atual. (Uma classe mais complexa de conta bancária poderia armazenar dados adicionais – talvez um número de conta, a taxa de juros paga, a data de envio do próximo extrato e assim por diante.)

Um objeto usa campos de instância para armazenar seu estado – os dados que ele precisa para executar seus métodos.

Um objeto armazena seus dados em *campos de instância*. Um *campo* é um termo técnico para uma posição de memória dentro de um bloco de memória. Uma *instância* de uma classe é um objeto dessa classe. Portanto, um campo de instância é uma posição de memória presente em cada objeto da classe.

A declaração de classe especifica os campos de instância:

```
public class BankAccount
{
   . . .
   private double balance;
}
```

Uma declaração de campo de instância consiste nas seguintes partes:

- Um *especificador de acesso* (normalmente `private`)
- O *tipo* do campo de instância (como `double`)
- O nome do campo de instância (como `balance`)

Cada objeto de uma classe tem seu próprio conjunto de campos de instância.

Você deve declarar todos os campos de instância como privados.

Cada objeto de uma classe tem seu próprio conjunto de campos de instância. Por exemplo, se `harrysChecking` e `momsSavings` forem dois objetos da classe `BankAccount`, então cada objeto terá um campo `balance` próprio, chamado `harrysChecking.balance` e `momsSavings.balance` (Veja Figura 5).

Os campos de instância geralmente são declarados com o especificador de acesso `private`. Esse especificador significa que os campos só podem ser acessados pelos métodos da *mesma classe* e por nenhum outro método. Por exemplo, a variável `balance` pode ser acessada pelo método `deposit` da classe `BankAccount`, mas não pelo método `main` de uma outra classe.

```
public class BankRobber
{
   public static void main(String[] args)
   {
      BankAccount momsSavings = new BankAccount(1000);
      . . .
      momsSavings.balance = -1000; // Erro
   }
}
```

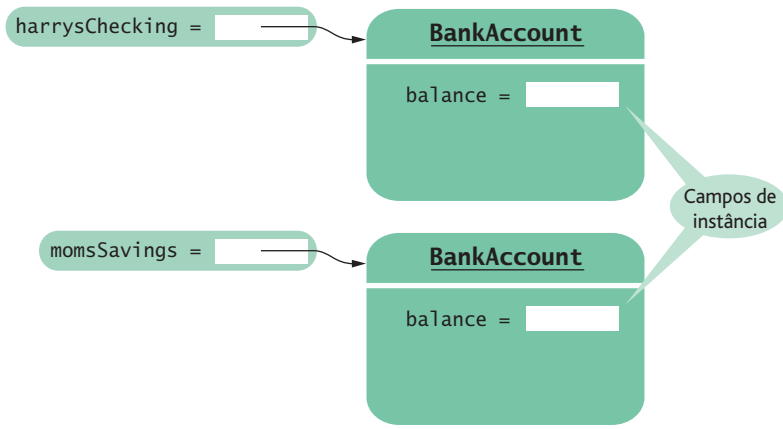


Figura 5 Campos de instância.

SINTAXE 3.4 Declaração de campo de instância

```
especificadorDeAcesso class NomeDaClasse
{
   . . .
   especificadorDeAcesso tipoDoCampo nomeDoCampo;
   . . .
}
```

Exemplo:

```
public class BankAccount
{
   . . .
   private double balance;
   . . .
}
```

Objetivo:

Definir um campo que estará presente em cada objeto da classe

O encapsulamento é o processo de ocultar dados de objeto e fornecer métodos para acessar estes dados.

Em outras palavras, se os campos de instância são declarados como privados, então todo o acesso aos dados deve ocorrer por meio dos métodos públicos. Portanto, os campos de instância de um objeto são efetivamente escondidos do programador que usa uma classe. Eles só têm importância para o programador que implementa a classe. O processo de ocultar os dados e fornecer métodos para o acesso a estes dados é chamado *encapsulamento*. Embora teoricamente seja possível que os campos de instância permaneçam públicos em Java, essa é uma prática bem incomum. Neste livro, os campos de instância sempre serão privados.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

7. Suponha que a classe `BankAccount` seja modificada para que cada conta bancária tenha um número de conta. Como essa alteração afeta os campos de instância?
8. Quais são os campos de instância da classe `Rectangle`?

3.5 Implementando construtores e métodos

Construtores contêm instruções para inicializar os campos de instância de um objeto.

Agora que definimos os campos de instância, vamos completar a classe `BankAccount` fornecendo os corpos dos construtores e dos métodos. Cada corpo contém uma seqüência de instruções. Iniciaremos com os construtores porque eles são muito simples. Um construtor tem uma tarefa simples: inicializar os campos de instância de um objeto.

Lembre-se de que projetamos a classe `BankAccount` para que ela tenha dois construtores. O primeiro construtor simplesmente configura o saldo como zero:

```
public BankAccount()
{
   balance = 0;
}
```

O segundo construtor configura o saldo de acordo com o valor fornecido como parâmetro de construção:

```
public BankAccount(double initialBalance)
{
   balance = initialBalance;
}
```

Para ver como esses construtores funcionam, vamos analisar a instrução

```
BankAccount harrysChecking = new BankAccount(1000);
```

um passo por vez. Eis os passos necessários quando a instrução é executada.

- Criar um novo objeto do tipo `BankAccount`.
- Chamar o segundo construtor (desde que um parâmetro de construção seja fornecido).
- Configurar a variável de parâmetro `initialBalance` como 1000.
- Configurar o campo de instância `balance` do objeto recém-criado como `initialBalance`.
- Retornar uma referência a objeto, isto é, a posição do objeto na memória, como o valor da expressão `new`.
- Armazenar essa referência a objeto na variável `harrysChecking`.

Vamos passar para a implementação do método `BankAccounts`. Eis o método `deposit`:

```
public void deposit(double amount)
{
   double newBalance = balance + amount;
   balance = newBalance;
}
```

Para entender exatamente o que esse método faz, considere esta instrução:

```
harrysChecking.deposit(500);
```

Essa instrução executa as seguintes ações:

- Configura a variável de parâmetro `amount` como 500.
- Busca o campo `balance` do objeto cuja localização está armazenada em `harrysChecking`.
- Adiciona o valor de `amount` a `balance` e armazena o resultado na variável `newBalance`.
- Armazena o valor de `newBalance` no campo de instância `balance`, sobrescrevendo o valor antigo.

O método `withdraw` é muito semelhante ao método `deposit`:

```
public void withdraw(double amount)
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Há apenas mais um método, `getBalance`. Diferentemente dos métodos `deposit` e `withdraw`, que modificam os campos de instância do objeto em que são invocados, o método `getBalance` retorna um valor de saída:

```
public double getBalance()
{
   return balance;
}
```

Utilize a instrução `return` para especificar o valor que um método retorna a quem o chamou.

A instrução `return` é uma instrução especial que indica que o método deve terminar e retornar uma saída à instrução que o chamou. No nosso caso, simplesmente retornamos o valor do campo de instância `balance`. Mais adiante veremos outros métodos que calculam e retornam expressões mais complexas.

Agora completamos a implementação da classe `BankAccount` – veja a listagem de código abaixo. Há apenas mais um passo: testar se a classe funciona corretamente. Esse é o tópico da próxima seção.

SINTAXE 3.5 A instrução `return`

`return` *expressão*;

ou

`return;`

Exemplo:

```
return balance;
```

Objetivo:

Especificar o valor que um método retorna e sair desse método imediatamente. O valor de retorno torna-se o valor da expressão da chamada do método.

ch03/account/BankAccount.java

```
/**
   Uma conta bancária tem um saldo que pode ser alterado por
   depósitos e saques.
*/
public class BankAccount
{
   /**
      Constrói uma conta bancária com saldo zero.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constrói uma conta bancária com saldo especificado.
      @param initialBalance saldo inicial
```

```
18      */
19      public BankAccount(double initialBalance)
20      {
21         balance = initialBalance;
22      }
23
24      /**
25         Deposita dinheiro na conta bancária.
26         @param amount valor a depositar
27      */
28      public void deposit(double amount)
29      {
30         double newBalance = balance + amount;
31         balance = newBalance;
32      }
33
34      /**
35         Retira dinheiro da conta bancária.
36         @param amount valor a retirar
37      */
38      public void withdraw(double amount)
39      {
40         double newBalance = balance - amount;
41         balance = newBalance;
42      }
43
44      /**
45         Obtém o saldo atual da conta bancária.
46         @return saldo atual
47      */
48      public double getBalance()
49      {
50         return balance;
51      }
52
53      private double balance;
54   }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

9. A classe `Rectangle` tem quatro campos de instância: `x`, `y`, `width` e `height`. Forneça uma possível implementação para o método `getWidth`.
10. Forneça uma possível implementação para o método `translate` da classe `Rectangle`.

COMO FAZER 3.1

Implementando uma classe

Esta é a primeira das várias seções "Como Fazer" deste livro. Usuários do sistema operacional Linux têm guias "como fazer" que fornecem respostas a questões comuns do tipo "Como começo?" e "O que eu faço em seguida?". Da mesma forma, as seções Como Fazer deste livro fornecem procedimentos passo a passo para executar tarefas específicas.

Freqüentemente você será solicitado a implementar uma classe. Por exemplo, uma lição de casa poderia solicitar para você implementar uma classe `CashRegister`.

Passo 1 Descubra quais métodos você foi encarregado de fornecer.

No exemplo da caixa registradora, você não terá de fornecer todos os recursos de uma caixa registradora real – há um número muito grande. O exercício deve informar *quais aspectos* de uma caixa registradora sua classe deve simular. Você deve ter recebido uma descrição, em linguagem simples, das operações que um objeto da sua classe deve executar, como esta:

- Registrar o preço de venda de um item adquirido.
- Inserir o valor do pagamento.
- Calcular o troco devido ao cliente.

Por simplicidade, estamos examinando uma caixa registradora muito simples. Um modelo mais sofisticado seria capaz de calcular o imposto sobre as vendas, o total das vendas diárias etc.

Passo 2 Especifique a interface pública.

Transforme a lista no Passo 1 em um conjunto de métodos, com tipos específicos para os parâmetros e para os valores de retorno. Muitos programadores acham que esse passo seria mais simples se escrevessem chamadas de método que poderiam ser aplicadas a um objeto de exemplo, como mostrado a seguir:

```
CashRegister register = new CashRegister();
register.recordPurchase(29.95);
register.recordPurchase(9.95);
register.enterPayment(50);
double change = register.giveChange();
```

Agora temos uma lista específica dos métodos.

- `public void recordPurchase(double amount)`
- `public void enterPayment(double amount)`
- `public double giveChange()`

Para completar a interface pública, você precisa especificar os construtores. Pense nas informações necessárias para criar um objeto desta classe. Às vezes, são desejáveis dois construtores: um que configura todos os campos com um padrão e outro que os configura com valores fornecidos pelo usuário.

No caso do exemplo da caixa registradora, um único construtor que cria um registro vazio é suficiente. Uma caixa registradora mais realista iniciaria com algumas moedas e cédulas para poder fornecer o troco exato, mas isso está além do escopo deste exercício.

Portanto, adicionaremos um único construtor:

- `public CashRegister()`

Passo 3 Documente a interface pública.

Eis a documentação, com comentários, que descreve a classe e seus métodos:

```
/**
   Uma caixa registradora soma as vendas e calcula o troco devido.
*/
public class CashRegister
{
   /**
      Constrói uma caixa registradora vazia (sem dinheiro).
   */
   public CashRegister()
   {
   }

   /**
      Registra a venda de um item.
      @param amount preço do item
   */
   public void recordPurchase(double amount)
   {
   }

   /**
      Insere o pagamento recebido do cliente.
      @param amount valor do pagamento
   */
   public void enterPayment(double amount)
   {
   }

   /**
      Calcula o troco devido e redefine a máquina para o próximo cliente.
      @return troco devido ao cliente
   */
   public double giveChange()
   {
   }
}
```

Passo 4 Determine os campos de instância.

Pense nas informações que um objeto precisa armazenar para fazer o trabalho. Lembre-se de que os métodos podem ser chamados em qualquer ordem! O objeto precisa ter memória interna suficiente para poder processar cada método utilizando apenas os campos de instância e os parâmetros de método. Analise cada método, talvez iniciando com um simples ou com um interessante, e pense no que é necessário para executar a tarefa dele. Faça os campos de instância armazenarem as informações que ele precisa.

No exemplo da caixa registradora, seria interessante manter um registro do valor total da compra e do valor do pagamento. Você pode calcular o troco devido a partir desses dois valores.

```
public class CashRegister
{
   ...
   private double purchase;
   private double payment
}
```

Passo 5 Implemente construtores e métodos.

Implemente os construtores e os métodos da sua classe, um por vez, iniciando com os mais fáceis. Por exemplo, eis a implementação do método `recordPurchase`:

```
public void recordPurchase(double amount)
{
   double newTotal = purchase + amount;
   purchase = newTotal;
}
```

Segue o método `giveChange`. Observe que esse método é um pouco mais sofisticado – ele calcula o troco devido e também redefine a caixa registradora para a próxima venda.

```
public double giveChange()
{
   double change = payment - purchase;
   purchase = 0;
   payment = 0;
   return change;
}
```

Se você achar que tem problemas com a implementação, talvez precise repensar sua escolha dos campos de instância. É comum que um iniciante comece com um conjunto de campos que pode não refletir exatamente o estado de um objeto. Não hesite em voltar atrás, adicionar ou modificar os campos.

Depois de concluir a implementação, compile sua classe e corrija todos os erros de compilador.

Passo 6 Teste sua classe.

Escreva um pequeno programa testador e o execute. O programa testador pode executar as chamadas de método encontradas no Passo 2.

```
public class CashRegisterTester
{
   public static void main(String[] args)
   {
      CashRegister register = new CashRegister();

      register.recordPurchase(29.50);
      register.recordPurchase(9.25);
      register.enterPayment(50);

      double change = register.giveChange();

      System.out.println(change);
      System.out.println("Expected: 11.25");
   }
}
```

A saída desse programa de teste é:

```
11.25
Expected: 11.25
```

Alternativamente, se você estiver usando um programa que permite testar objetos interativamente, como o BlueJ, construa um objeto e aplique as chamadas de método.

3.6 Teste de unidade

Na seção anterior, completamos a implementação da classe `BankAccount`. O que você pode fazer com ela? É claro que pode compilar o arquivo `BankAccount.java`, mas você não pode *executar* o arquivo `BankAccount.class` resultante. Ele não contém um método `main`. Isso é normal – a maioria das classes não contém um método `main`.

Um teste de unidade verifica se uma classe funciona corretamente de forma isolada, fora de um programa completo.

Com o tempo, sua classe pode tornar-se parte de um programa maior que interage com usuários, armazena dados em arquivos e assim por diante. Entretanto, antes de integrar uma classe a um programa, sempre é uma boa idéia testá-lo isoladamente. O teste isolado, fora de um programa completo, é chamado *teste de unidade*.

Para testar sua classe, há duas opções. Alguns ambientes de desenvolvimento interativos têm comandos para construir objetos e invocar métodos (veja o Tópico Avançado 2.1). Você pode então testar uma classe simplesmente criando um objeto, chamando métodos e verificando se você obtém os valores de retorno esperados. A Figura 6 mostra o resultado da chamada do método `getBalance` a partir de um objeto `BankAccount` no BlueJ.

Para testar uma classe, utilize um ambiente de teste interativo ou escreva uma classe testadora para executar as instruções de teste.

Alternativamente, você pode escrever uma *classe testadora*. Uma classe testadora é uma classe com um método `main` que contém instruções para executar métodos de outra classe. Uma classe testadora em geral executa as seguintes ações:

1. Cria um ou mais objetos da classe que está sendo testada.
2. Invoca um ou mais métodos.
3. Imprime um ou mais resultados.
4. Imprime os resultados esperados.

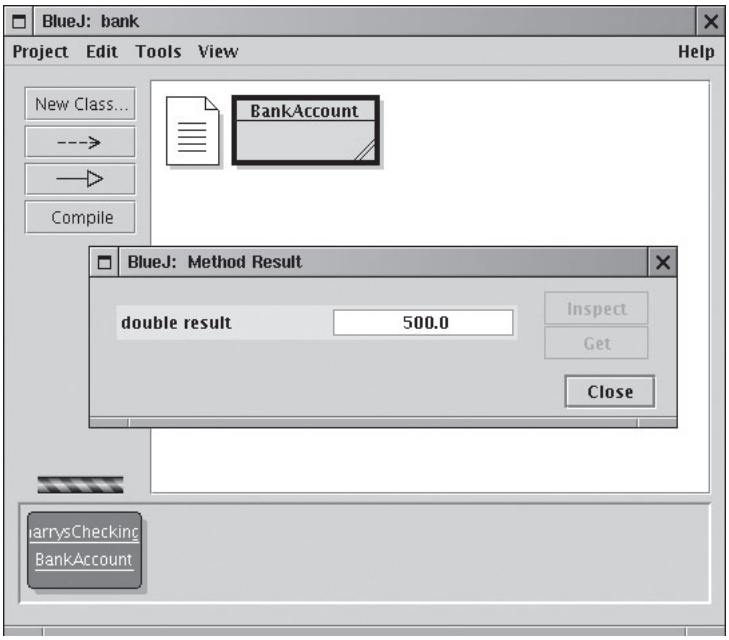


Figura 6
Valor de retorno do método `getBalance` no BlueJ.

A classe `MoveTester` na Seção 2.8 é um bom exemplo de uma classe testadora. Essa classe executa os métodos da classe `Rectangle` – uma classe da biblioteca Java.

Eis uma classe para executar os métodos da classe `BankAccount`. O método `main` cria um objeto do tipo `BankAccount`, invoca os métodos `deposit` e `withdraw` e então exibe o saldo restante na console.

Também imprimimos o valor que esperamos ver. No nosso programa de exemplo, depositamos US$ 2.000 e sacamos US$ 500. Portanto esperamos um saldo de US$ 1.500.

ch03/account/BankAccountTester.java

```
1  /**
2     Uma classe para testar a classe BankAccount.
3  */
4  public class BankAccountTester
5  {
6     /**
7        Testa os métodos da classe BankAccount.
8        @param args não utilizado
9     */
10    public static void main(String[] args)
11    {
12       BankAccount harrysChecking = new BankAccount();
13       harrysChecking.deposit(2000);
14       harrysChecking.withdraw(500);
15       System.out.println(harrysChecking.getBalance());
16       System.out.println("Expected: 1500");
17    }
18 }
```

Saída

```
1500
Expected: 1500
```

Para criar um programa, você precisa combinar as classes `BankAccount` e `BankAccountTester`. Os detalhes da criação do programa dependem do compilador e do ambiente de desenvolvimento. Na maioria dos ambientes, você precisa seguir estes passos:

1. Criar uma nova subpasta para seu programa.
2. Criar dois arquivos, um para cada classe.
3. Compilar os dois arquivos.
4. Executar o programa de teste.

Muitos estudantes se surpreendem com o fato de um programa tão simples conter duas classes. Mas isso é normal. As duas classes têm propósitos completamente diferentes. A classe `BankAccount` descreve os objetos que calculam os saldos bancários. A classe `BankAccountTester` testa um objeto `BankAccount`.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

11. Quando você executa o programa `BankAccountTester`, quantos objetos da classe `BankAccount` são criados? Quantos objetos do tipo `BankAccountTester`?
12. Por que a classe `BankAccountTester` é desnecessária em ambientes de desenvolvimento que permitem testes interativos, como o BlueJ?

DICA DE PRODUTIVIDADE 3.2

Utilizando a linha de comando de maneira eficiente

A Dica de Produtividade 3.2 discute como utilizar de forma eficiente a linha de comando. Leia essa dica se você costuma carregar programas Java a partir da linha de comando em vez de usar ambiente de desenvolvimento integrado.

3.7 Categorias de variáveis

Fechamos este capítulo com duas seções de natureza mais técnica, examinando variáveis e parâmetros em detalhes.

Vimos três diferentes categorias de variáveis neste capítulo:

1. *Campos de instância* (às vezes chamados de *variáveis de instância*), como a variável `balance` da classe `BankAccount`.
2. *Variáveis locais*, como a variável `newBalance` do método `deposit`.
3. *Variáveis de parâmetro*, como a variável `amount` do método `deposit`.

Campos de instância pertencem a objetos. Variáveis de parâmetro e variáveis locais pertencem a um método – elas são excluídas quando o método é encerrado.

Essas variáveis são semelhantes em um aspecto – todas armazenam valores de tipos específicos. Mas há diferenças importantes. A primeira delas é o *tempo de vida*.

Um campo de instância pertence a um objeto. Cada objeto tem sua própria cópia de cada campo de instância. Por exemplo, se você tiver dois objetos `BankAccounts` (digamos, `harrysChecking` e `momsSavings`), então cada um deles terá um campo `balance` próprio. Quando um objeto é construído, seus campos de instância são criados. Os campos permanecem vivos até que nenhum método o utilize mais. (A máquina virtual Java contém um agente chamado *coletor de lixo* que periodicamente faz uma coleta dos objetos não mais utilizados.)

Variáveis locais e variáveis de parâmetro pertencem a um método. Quando o método é executado, essas variáveis ganham vida. Quando o método é concluído, elas morrem imediatamente (veja Figura 7).

Por exemplo, se você chamar

```
harrysChecking.deposit(500); ❶
```

uma variável de parâmetro chamada `amount` é criada e inicializada com o valor de parâmetro 500. Quando o método retorna, a variável `amount` morre. O mesmo é válido para a variável local `newBalance`. Quando o método `deposit` alcança a linha

```
double newBalance = balance + amount; ❷
```

a variável ganha vida e é inicializada com a soma do saldo do objeto e o valor do depósito. O tempo de vida dessa variável se estende até o final do método.

Mas o método `deposit` tem um último efeito. Sua próxima linha,

```
balance = newBalance;
```
3

configura o campo de instância `balance`, que permanece vivo depois do final do método `deposit`, pelo tempo que o objeto `BankAccount` estiver em uso.

Campos de instância são inicializados com um valor padrão, mas você precisa inicializar as variáveis locais.

A segunda diferença importante entre campos de instância e variáveis locais é em relação à *inicialização*. Você deve inicializar todas as variáveis locais. Se você não inicializar uma variável local, o compilador reclamará quando você tentar utilizá-la.

Variáveis de parâmetro são inicializadas com os valores fornecidos na chamada do método.

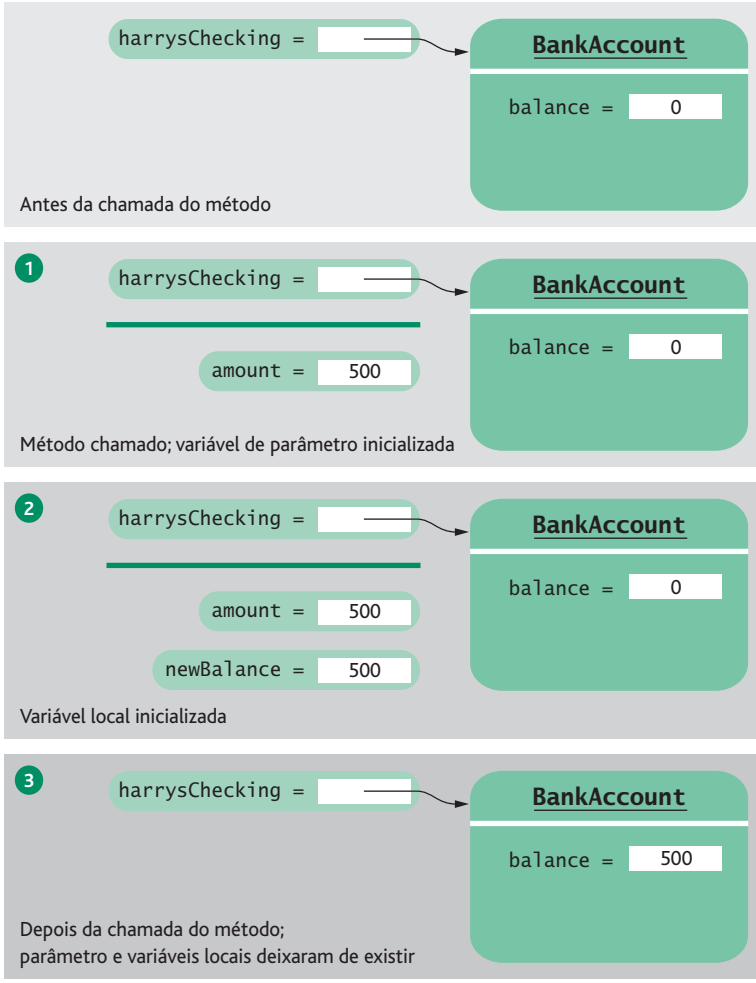


Figura 7 Tempo de vida das variáveis.

Campos de instância são inicializados com um valor padrão se você não os configurar explicitamente em um construtor. Campos de instância numéricos são inicializados como 0. Referências a objetos são configuradas com um valor especial chamado `null`. Se uma referência a objeto for `null`, então ela não referenciará nenhum objeto. Discutiremos o valor `null` mais detalhadamente na Seção 5.2.5. Inicialização inadvertida com 0 ou `null` é uma causa comum de erros. Portanto, é uma questão de bom estilo inicializar *cada* campo de instância explicitamente em cada construtor.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

13. O que variáveis locais e variáveis de parâmetro têm em comum? Em que aspecto essencial elas diferem?
14. Durante a execução do programa `BankAccountTester` na seção anterior, quantos campos de instância, variáveis locais e variáveis de parâmetro foram criados e quais eram seus nomes?

ERRO COMUM 3.1

Esquecendo de inicializar referências a objetos em um construtor

Assim como é um erro comum esquecer de inicializar uma variável local, é fácil esquecer dos campos de instância. Cada construtor precisa garantir que todos os campos de instância são configurados com valores apropriados.

Se você não inicializar um campo de instância, o compilador Java o inicializará para você. Números são inicializados com 0, mas referências a objeto – como variáveis alfanuméricas – são configuradas com a referência `null`.

Naturalmente, com freqüência 0 é um padrão conveniente para números. Entretanto, `null` quase nunca é um padrão conveniente para objetos. Considere esse construtor "preguiçoso" para uma versão modificada da classe `BankAccount`:

```
public class BankAccount
{
   public BankAccount() {} // Nenhuma instrução
   . . .
   private double balance;
   private String owner;
}
```

O `balance` é configurado como 0 e o campo `owner` é configurado com uma referência `null`. Isso é um problema – não é permitido chamar métodos a partir da referência `null`.

Se você esquecer de inicializar uma variável *local* em um *método*, o compilador sinalizará isso como um erro e será necessário corrigi-lo antes de o programa executar. Se cometer o mesmo erro com um campo de *instância* em uma classe, o compilador fornecerá uma inicialização padrão e o erro só se tornará aparente quando o programa for executado.

Para evitar esse problema, torne um hábito inicializar cada campo de instância em cada construtor.

3.8 Parâmetros de método implícitos e explícitos

Na Seção 2.4, você aprendeu que um método tem um parâmetro implícito – o objeto em que o método é invocado – e parâmetros explícitos, que são incluídos entre parênteses. Nesta seção, examinaremos esses parâmetros mais detalhadamente.

Examine uma invocação particular do método `deposit`:

```
momsSavings.deposit(500);
```

Agora examine esse código do método `deposit` mais uma vez:

```
public void deposit(double amount)
{
   double newBalance = balance + amount;
   balance = newBalance;
}
```

A variável de parâmetro `amount` é configurada como 500 quando o método `deposit` é inicializado. Mas o que `balance` significa exatamente? Afinal de contas, nosso programa pode ter múltiplos objetos `BankAccount`, e *cada um deles* pode conter um saldo próprio.

Naturalmente, como depositamos o dinheiro em `momsSavings`, o saldo deve significar `momsSavings.balance`. Em geral, quando você referencia um campo de instância dentro de um método, isso significa o campo de instância do objeto a partir do qual o método foi chamado.

O parâmetro implícito de um método é o objeto em que o método é invocado. A referência `this` denota o parâmetro implícito.

Portanto, a chamada ao método `deposit` depende de dois valores: o objeto que `momsSavings` referencia e o valor `500`. O parâmetro `amount` dentro de parênteses é chamado de parâmetro *explícito*, porque ele recebe explicitamente um nome na definição do método. Mas a referência ao objeto conta bancária não é explícita na definição do método – ela é chamada de *parâmetro implícito* do método.

Se necessário, você pode acessar o parâmetro implícito – o objeto em que o método foi chamado – com a palavra-chave `this`. Por exemplo, na invocação do método anterior, `this` foi configurado como `momsSavings` e `amount` como 500 (veja Figura 8).

Cada método tem um parâmetro implícito. Você não atribui um nome ao parâmetro implícito. Ele sempre se chama `this`. (Há uma exceção à regra de que cada método tem um parâmetro implícito: métodos `static` não têm. Iremos discuti-los no Capítulo 8.) Em contraposição, métodos podem ter qualquer quantidade de parâmetros explícitos – os quais você pode nomear da maneira que preferir – ou simplesmente nenhum parâmetro explícito.

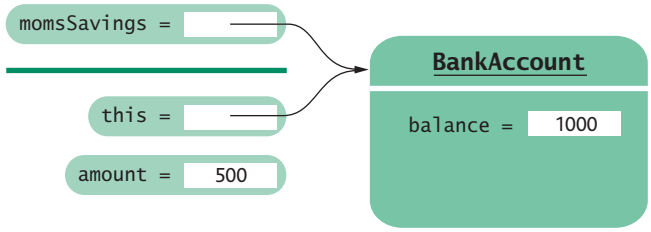


Figura 8 Parâmetro implícito de uma chamada de método.

Em seguida, examine atentamente a implementação do método `deposit`. A instrução

```
double newBalance = balance + amount;
```

na verdade significa:

```
double newBalance = this.balance + amount;
```

O uso de um nome de campo de instância em um método indica o campo de instância do parâmetro implícito.

Quando você referencia um campo de instância em um método, o compilador aplica-o automaticamente ao parâmetro `this`. Alguns programadores preferem inserir o parâmetro `this` manualmente antes de cada campo de instância porque acham que isso torna o código mais claro. Eis um exemplo:

```
public void deposit(double amount)
{
   double newBalance = this.balance + amount;
   this.balance = newBalance;
}
```

Talvez você queira experimentar e ver se gosta desse estilo.

Já vimos como utilizar objetos e implementar classes, e você aprendeu alguns detalhes técnicos importantes sobre variáveis e parâmetros de método. No próximo capítulo, veremos outros detalhes sobre os tipos de dados mais fundamentais da linguagem Java.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

15. Quantos parâmetros implícitos e explícitos o método `withdraw` da classe `BankAccount` tem e quais são seus nomes e tipos?
16. No método `deposit`, qual é o significado de `this.amount`? Ou, se a expressão não tiver significado, por que não?
17. Quantos parâmetros implícitos e explícitos o método `main` da classe `BankAccountTester` tem e como eles são chamados?

ERRO COMUM 3.2

Tentando chamar um método sem um parâmetro implícito

Suponha que seu método `main` contém a instrução:

```
withdraw(30); // Erro
```

O compilador não saberá qual conta acessar para retirar o dinheiro. Você precisa fornecer uma referência a um objeto do tipo `BankAccount`:

```
BankAccount harrysChecking = new BankAccount();
harrysChecking.withdraw(30);
```

Mas há uma situação em que é legítimo invocar um método sem, aparentemente, um parâmetro implícito. Considere a seguinte modificação na classe `BankAccount`. Adicione um método para aplicar a tarifa da conta mensal:

```
public class BankAccount
{  . . .
   public void monthlyFee()
   {
```

```
        withdraw(10); // Retira US$ 10 dessa conta
      }
   }
```

Isso significa retirar do mesmo objeto conta bancária que está executando a operação `monthlyFee`. Em outras palavras, o parâmetro implícito do método `withdraw` é o parâmetro implícito (invisível) do método `monthlyFee`.

Se você achar confuso ter um parâmetro invisível, sempre será possível utilizar o parâmetro `this` para tornar a leitura desse método mais fácil:

```
public class BankAccount
{  . . .
   public void monthlyFee()
   {
      this.withdraw(10); // Retira US$ 10 dessa conta
   }
}
```

Tópico avançado 3.1

Chamando um construtor a partir de um outro

O Tópico Avançado 3.1 descreve como você pode minimizar código comum em múltiplos construtores com a palavra-chave `this` para chamar um construtor a partir de um outro.

Fato aleatório 3.1

Urnas eletrônicas

O Fato Aleatório 3.1 discute as questões que surgem no projeto de urnas eletrônicas. Muitos cientistas da computação acreditam que as urnas eletrônicas devem ser complementadas por uma auditoria em papel verificável pelo eleitor, porque é impossível, com a tecnologia atual, afirmar que softwares estão livres de erros e que não foram adulterados.

3.9 Classes de formas gráficas

Continuamos com a trilha opcional sobre elementos gráficos discutindo como organizar desenhos complexos de uma maneira mais orientada a objetos. Sinta-se livre para pular esta seção se não estiver interessado em aplicações gráficas.

Uma boa idéia é criar uma classe para cada parte de um desenho que possa ser utilizado mais de uma vez.

Quando você cria um desenho composto de partes complexas, como o da Figura 9, uma boa idéia é criar uma classe separada para cada parte. Forneça um método `draw` que desenhe a forma e forneça um construtor para configurar a posição dela. Por exemplo, eis o esboço da classe `Car`.

```
public class Car
{
   public Car(int x, int y)
   {
```

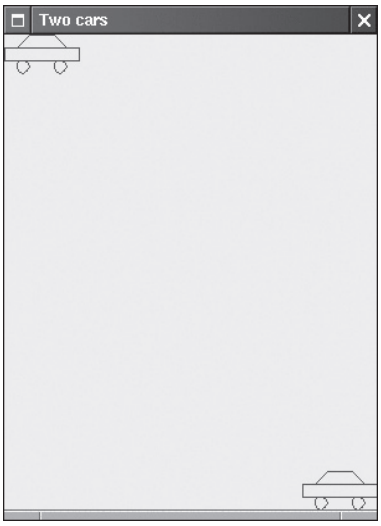


Figura 9
O componente carro desenha duas vezes a forma do carro.

```
      // Lembra a posição
      . . .
   }
   public void draw(Graphics2D g2)
   {
      // Instruções de desenho
      . . .
   }
}
```

Para entender como desenhar uma forma complexa, crie o esboço em um papel quadriculado.

Você encontrará a definição completa desta classe no final desta seção. O método `draw` contém uma seqüência bastante longa de instruções para desenhar a carroceria, o teto e os pneus.

As coordenadas das peças do carro parecem um pouco arbitrárias. Para conseguir valores adequados, desenhe a imagem no papel quadriculado e anote as coordenadas (Figura 10).

O programa que produz a Figura 9 é composto de três classes.

- A classe `Car` é responsável por desenhar um único carro. Dois objetos dessa classe são construídos, um para cada carro.
- A classe `CarComponent` exibe o desenho.
- A classe `CarViewer` mostra um frame que contém um `CarComponent`.

Vamos examinar mais atentamente a classe `CarComponent`. O método `paintComponent` desenha dois carros. Colocamos um carro no canto superior esquerdo da janela e o outro na parte inferior direita. Para calcular a posição inferior direita, chamamos os métodos `getWidth` e `getHeight` da classe `JComponent`. Esses métodos retornam as dimensões do componente. Subtraímos as dimensões do carro:

```
Car car1 = new Car(0, 0);

int x = getWidth() - 60;
int y = getHeight() - 30;
Car car2 = new Car(x, y);
```

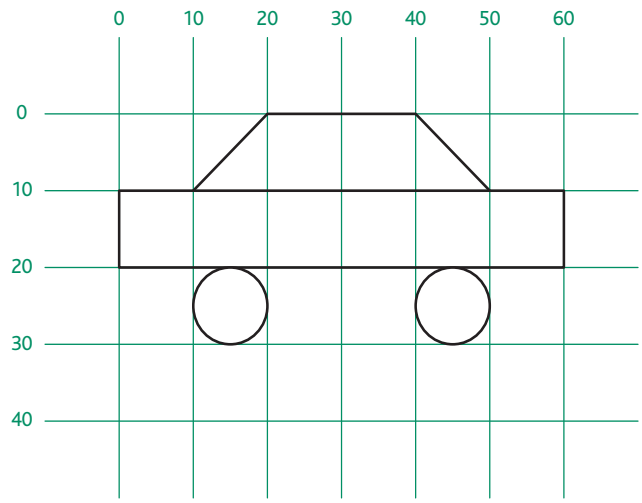


Figura 10 Utilizando papel quadriculado para encontrar as coordenadas da forma.

Preste bastante atenção à chamada a `getWidth` dentro do método `paintComponent` de `CarComponent`. A chamada de método não tem parâmetro implícito, o que significa que o método é aplicado ao mesmo objeto que executa o método `paintComponent`. O componente simplesmente obtém *sua própria* largura.

Execute o programa e redimensione a janela. Observe que o segundo carro sempre acaba no canto inferior direito da janela. Sempre que a janela é redimensionada, o método `paintComponent` é chamado e a posição de carro é recalculada levando em consideração as dimensões atuais do componente.

ch03/car/CarComponent.java

```
1  import java.awt.Graphics;
2  import java.awt.Graphics2D;
3  import javax.swing.JComponent;
4
5  /**
6     Esse componente desenha duas formas do carro.
7  */
8  public class CarComponent extends JComponent
9  {
10    public void paintComponent(Graphics g)
11    {
12       Graphics2D g2 = (Graphics2D) g;
13
14       Car car1 = new Car(0, 0);
15
16       int x = getWidth() - 60;
17       int y = getHeight() - 30;
18
19       Car car2 = new Car(x, y);
20
```

```
21         car1.draw(g2);
22         car2.draw(g2);
23      }
24   }
```

ch03/car/Car.java

```
1   import java.awt.Graphics2D;
2   import java.awt.Rectangle;
3   import java.awt.geom.Ellipse2D;
4   import java.awt.geom.Line2D;
5   import java.awt.geom.Point2D;
6
7   /**
8      Uma forma do carro que pode ser posicionada em qualquer lugar na tela.
9   */
10  public class Car
11  {
12     /**
13        Constrói um carro com o canto superior esquerdo fornecido.
14        @param x coordenada x do canto superior esquerdo
15        @param y coordenada y do canto superior esquerdo
16     */
17     public Car(int x, int y)
18     {
19        xLeft = x;
20        yTop = y;
21     }
22
23     /**
24        Desenha o carro.
25        @param g2 contexto gráfico
26     */
27     public void draw(Graphics2D g2)
28     {
29        Rectangle body
30              = new Rectangle(xLeft, yTop + 10, 60, 10);
31        Ellipse2D.Double frontTire
32              = new Ellipse2D.Double(xLeft + 10, yTop + 20, 10, 10);
33        Ellipse2D.Double rearTire
34              = new Ellipse2D.Double(xLeft + 40, yTop + 20, 10, 10);
35
36        // Parte inferior do pára-brisa frontal
37        Point2D.Double r1
38              = new Point2D.Double(xLeft + 10, yTop + 10);
39        // Parte frontal do teto
40        Point2D.Double r2
41              = new Point2D.Double(xLeft + 20, yTop);
42        // Parte traseira do teto
43        Point2D.Double r3
44              = new Point2D.Double(xLeft + 40, yTop);
45        // Parte inferior do pára-brisa traseiro
46        Point2D.Double r4
47              = new Point2D.Double(xLeft + 50, yTop + 10);
```

```
48
49         Line2D.Double frontWindshield
50               = new Line2D.Double(r1, r2);
51         Line2D.Double roofTop
52               = new Line2D.Double(r2, r3);
53         Line2D.Double rearWindshield
54               = new Line2D.Double(r3, r4);
55
56         g2.draw(body);
57         g2.draw(frontTire);
58         g2.draw(rearTire);
59         g2.draw(frontWindshield);
60         g2.draw(roofTop);
61         g2.draw(rearWindshield);
62      }
63
64      private int xLeft;
65      private int yTop;
66   }
```

ch03/car/CarViewer.java

```
1   import javax.swing.JFrame;
2
3   public class CarViewer
4   {
5      public static void main(String[] args)
6      {
7         JFrame frame = new JFrame();
8
9         frame.setSize(300, 400);
10        frame.setTitle("Two cars");
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12
13        CarComponent component = new CarComponent();
14        frame.add(component);
15
16        frame.setVisible(true);
17     }
18  }
```

Autoverificação da aprendizagem

18. Qual classe precisa ser modificada para que os dois carros possam ser posicionados um ao lado do outro?

19. Qual classe precisa ser modificada para que os pneus do carro possam ser pintados em preto e qual modificação deve ser feita?

20. Como você faz para dobrar o tamanho dos carros?

COMO FAZER 3.2

Desenhando formas gráficas

Você pode escrever programas que exibem uma ampla variedade de formas gráficas. Estas instruções fornecem um procedimento passo a passo para decompor um desenho em partes e implementar um programa que produz o desenho.

Passo 1 Determine as formas que você precisa para o desenho.

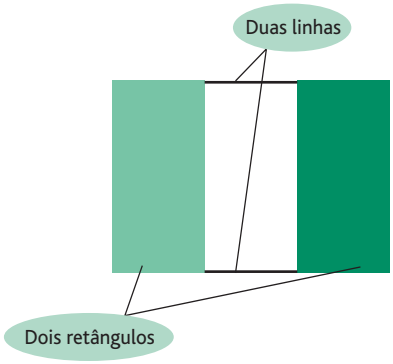
Você pode utilizar as seguintes formas:

- Quadrados e retângulos
- Círculos e elipses
- Linhas

O contorno dessas formas pode ser desenhado com uma cor e preenchido com outra. Você também pode utilizar texto para rotular partes do seu desenho.

O design de algumas bandeiras nacionais consiste em três seções de igual largura e de diferentes cores, lado a lado:

Você poderia desenhar esse tipo de bandeira utilizando três retângulos. Mas se, por exemplo, o retângulo do meio for branco, como é na bandeira da Itália (verde, branco e vermelho), é mais fácil e dá uma aparência melhor desenhar uma linha na parte superior e outra na parte inferior da parte central:



Passo 2 Encontrar as coordenadas para as formas.

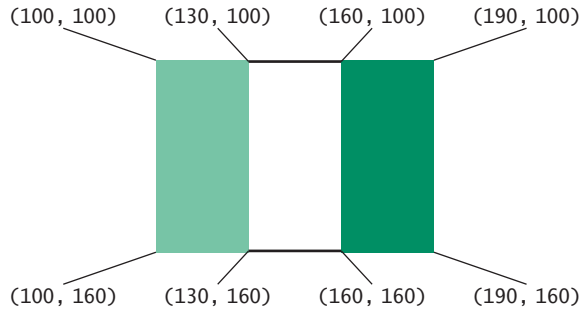
Agora você precisa encontrar as posições exatas para as formas geométricas.

- Para os retângulos, você precisa das posições *x* e *y* do canto superior esquerdo, da largura e da altura.
- Para as elipses, você precisa do canto superior esquerdo, da largura e da altura do retângulo delimitador.
- Para as linhas, você precisa das posições *x* e *y* do ponto inicial e do ponto final.
- Para o texto, você precisa das posições *x* e *y* do ponto de base.

Um tamanho comumente utilizado para uma janela é 300 por 300 pixels. Você não quer que a bandeira fique totalmente comprimida contra a parte superior, portanto, talvez o canto superior esquerdo da bandeira devesse estar no ponto (100, 100).

Muitas bandeiras, como a bandeira da Itália, têm uma proporção entre a largura e a altura de 3:2. (Quase sempre você pode encontrar as proporções exatas para uma bandeira específica fazendo uma pequena pesquisa na Internet em um dos vários sites do tipo "Bandeiras do Mundo".) Por exemplo, se criar a bandeira com uma largura de 90 pixels, então ela deverá ter uma altura de 60 pixels. (Por que não uma largura de 100 pixels? A altura então seria $100 \cdot 2 / 3 \approx 67$, o que parece mais estranho.)

Agora você pode calcular as coordenadas de todos os pontos importantes da forma:



Passo 3 Escreva instruções Java para desenhar as formas.

No nosso exemplo, há dois retângulos e duas linhas:

```
Rectangle leftRectangle
      = new Rectangle(100, 100, 30, 60);
Rectangle rightRectangle
      = new Rectangle(160, 100, 30, 60);
Line2D.Double topLine
      = new Line2D.Double(130, 100, 160, 100);
Line2D.Double bottomLine
      = new Line2D.Double(130, 160, 160, 160);
```

Se for mais ambicioso, você poderá expressar as coordenadas como algumas variáveis. No caso da bandeira, escolhemos arbitrariamente o canto superior esquerdo e a largura. Todas as outras coordenadas seguem essas escolhas. Se decidir seguir a abordagem ambiciosa, os retângulos e as linhas serão determinados desta maneira:

```
Rectangle leftRectangle = new Rectangle(
      xLeft, yTop,
      width / 3, width * 2 / 3);
Rectangle rightRectangle = new Rectangle(
      xLeft + 2 * width / 3, yTop,
      width / 3, width * 2 / 3);
```

```
Line2D.Double topLine = new Line2D.Double(
      xLeft + width / 3, yTop,
      xLeft + width * 2 / 3, yTop);
Line2D.Double bottomLine = new Line2D.Double(
      xLeft + width / 3, yTop + width * 2 / 3,
      xLeft + width * 2 / 3, yTop + width * 2 / 3);
```

Agora você precisa preencher os retângulos e desenhar as linhas. Para a bandeira da Itália, o retângulo esquerdo é verde e o direito é vermelho. Lembre-se de mudar as cores antes das operações de desenho e preenchimento:

```
g2.setColor(Color.GREEN);
g2.fill(leftRectangle);
g2.setColor(Color.RED);
g2.fill(rightRectangle);
g2.setColor(Color.BLACK);
g2.draw(topLine);
g2.draw(bottomLine);
```

Passo 4 Combine as instruções de desenho com o código básico do componente.

```
public class MyComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      // Seu código de desenho entra aqui
      . . .
   }
}
```

No nosso exemplo, você pode simplesmente adicionar todas as formas e instruções de desenho dentro do método `paintComponent`:

```
public class ItalianFlagComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      Rectangle leftRectangle
            = new Rectangle(100, 100, 30, 60);
      . . .
      g2.setColor(Color.GREEN);
      g2.fill(leftRectangle);
      . . .
   }
}
```

Essa abordagem é aceitável para desenhos simples, mas não é muito orientada a objetos. Afinal de contas, uma bandeira é um objeto. É melhor criar uma classe separada para bandeira. Você pode então desenhar diferentes bandeiras em diferentes posições e tamanhos. Especifique os tamanhos em um construtor e forneça um método `draw`:

```
public class ItalianFlag
{
   public ItalianFlag(double x, double y, double aWidth)
   {
     xLeft = x;
```

```
      yTop = y;
      width = aWidth;
   }

   public void draw(Graphics2D g2)
   {
      Rectangle leftRectangle = new Rectangle(
            xLeft, yTop,
            width / 3, width * 2 / 3);
      . . .
      g2.setColor(Color.GREEN);
      g2.fill(leftRectangle);
      . . .
   }

   private int xLeft;
   private int yTop;
   private double width;
}
```

Você ainda precisará de uma classe separada para o componente, mas ela é bem simples:

```
public class ItalianFlagComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      ItalianFlag flag = new ItalianFlag(100, 100, 90);
      flag.draw(g2);
   }
}
```

Passo 5 Escreva a classe de visualizadores.

Forneça uma classe de visualizadores, com um método `main` em que você cria um frame, adiciona seu componente e torna seu frame visível. A classe de visualizadores é pura rotina; você só precisa alterar uma única linha para mostrar um componente diferente.

```
import javax.swing.*;

public class ItalianFlagViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      frame.setSize(300, 400);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      ItalianFlagComponent component = new ItalianFlagComponent();
      frame.add(component);

      frame.setVisible(true);
   }
}
```

FATO ALEATÓRIO 3.2

Computação gráfica

O Fato Aleatório 3.2 discute computação gráfica, a tecnologia utilizada para gerar e manipular imagens visuais em um computador.

RESUMO DO CAPÍTULO

1. Para implementar uma classe, primeiro é necessário saber quais métodos são necessários.
2. Uma definição de método contém um especificador de acesso (normalmente `public`), um tipo de retorno, um nome de método, parâmetros e o corpo do método.
3. Construtores contêm instruções para inicializar objetos. O nome do construtor sempre é o mesmo que o nome da classe.
4. Utilize os comentários da documentação para descrever as classes e os métodos públicos dos seus programas.
5. Forneça comentários de documentação para cada classe, cada método, cada parâmetro e cada valor de retorno.
6. Um objeto usa campos de instância para armazenar seu estado – os dados que ele precisa para executar seus métodos.
7. Cada objeto de uma classe tem seu próprio conjunto de campos de instância.
8. Você deve declarar todos os campos de instância como privados.
9. O encapsulamento é o processo de ocultar dados de objeto e fornecer métodos para o acesso a dados.
10. Construtores contêm instruções para inicializar os campos de instância de um objeto.
11. Utilize a instrução `return` para especificar o valor que um método retorna para seu chamador.
12. Um teste de unidade verifica se uma classe funciona corretamente quando isolada, ou seja, fora de um programa completo.
13. Para testar uma classe, utilize um ambiente de teste interativo ou escreva uma classe testadora para executar as instruções de teste.
14. Campos de instância pertencem a um objeto. Variáveis de parâmetro e variáveis locais pertencem a um método – elas deixam de existir quando o método é concluído.
15. Campos de instância são inicializados com um valor padrão, mas as variáveis locais devem ser inicializadas explicitamente.
16. O parâmetro implícito de um método é o objeto a partir do qual o método é invocado. A referência `this` indica o parâmetro implícito.
17. O uso de um nome de campo de instância em um método indica o campo de instância do parâmetro implícito.
18. Uma boa idéia é criar uma classe para cada parte de um desenho que possa ocorrer mais de uma vez.
19. Para entender como desenhar uma forma gráfica complexa, crie o esboço em um papel quadriculado.

EXERCÍCIOS DE REVISÃO

★ **Exercício R3.1.** Por que o construtor `BankAccount(double initialBalance)` não é estritamente necessário?

★ **Exercício R3.2.** Explique a diferença entre:

```
BankAccount b;
```

e

```
BankAccount b = new BankAccount(5000);
```

★ **Exercício R3.3.** Explique a diferença entre:

```
new BankAccount(5000);
```

e

```
BankAccount b = new BankAccount(5000);
```

★ **Exercício R3.4.** O que acontece na nossa implementação da classe `BankAccount` quando mais dinheiro é sacado da conta além daquele no saldo atual?

★ **Exercício R3.5.** Qual é o valor retornado a partir da chamada do método `b.getBalance()` depois destas operações?

```
BankAccount b = new BankAccount(10);
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
```

★★ **Exercício R3.6.** Se `b1` e `b2` referenciarem objetos da classe `BankAccount`, considere as instruções a seguir.

```
b1.deposit(b2.getBalance());
b2.deposit(b1.getBalance());
```

Agora os saldos de `b1` e `b2` são idênticos? Explique.

★★ **Exercício R3.7.** O que é a referência `this`? Por que você a utilizaria?

★★ **Exercício R3.8.** O que o seguinte método faz? Dê um exemplo de como você pode chamar o método.

```
public class BankAccount
{
   public void mystery(BankAccount that, double amount)
   {
      this.balance = this.balance - amount;
      that.balance = that.balance + amount;
   }
   . . . // Outros métodos de conta bancária
}
```

★★ **Exercício R3.9.** Suponha que você queira implementar uma classe `TimeDepositAccount`. Uma conta de depósito que, após um tempo especificado, recebe uma taxa fixa de juros que deve ser definida no construtor, juntamente com o saldo inicial. Forneça um método para obter o saldo atual. Forneça um método para adicionar os juros ganhos à conta. Esse método não deve ter nenhum parâmetro porque a taxa de juros já é conhecida. Ele

também não deve ter nenhum valor de retorno porque você já forneceu um método para obter o saldo atual. Não é possível depositar quantias adicionais nessa conta. Forneça um método `withdraw` que remove todo o saldo. Saques parciais não são permitidos.

★★ **Exercício R3.10.** Quais são os métodos modificadores e os métodos de acesso da classe `CashRegister`?

★ **Exercício R3.11.** Explique a diferença entre uma variável local e uma variável de parâmetro.

★ **Exercício R3.12.** Explique a diferença entre um campo de instância e uma variável local.

★★G **Exercício R3.13.** Suponha que você queira estender o programa visualizador de carros da Seção 3.9 para exibir uma cena suburbana com vários carros e casas. De que classes você precisa?

★★★G **Exercício R3.14.** Explique por que as chamadas aos métodos `getWidth` e `getHeight` na classe `CarComponent` não têm nenhum parâmetro explícito.

★★G **Exercício R3.15.** Como você modificaria a classe `Car` para mostrar carros de diferentes tamanhos?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P3.1.** Escreva uma classe `BankAccountTester` cujo método `main` cria uma conta bancária, deposita US$ 1.000, retira US$ 500, retira mais US$ 400 e então imprime o saldo remanescente. Imprima também o resultado esperado.

★ **Exercício P3.2.** Adicione um método:

```
public void addInterest(double rate)
```

à classe `BankAccount` que adiciona uma dada taxa de juros. Por exemplo, depois das instruções:

```
BankAccount momsSavings = new BankAccount(1000);
momsSavings.addInterest(10); // 10% de juros
```

o saldo em `momsSavings` é US$ 1.100. Também forneça uma classe `BankAccountTester` que imprime o saldo real e o esperado.

★★ **Exercício P3.3.** Escreva uma classe `SavingsAccount` semelhante à classe `BankAccount`, exceto que pelo fato de conter um campo de instância `interest` extra. Forneça um construtor que configure tanto o saldo inicial como a taxa de juros. Forneça um método `addInterest` (sem parâmetro explícito) que adiciona juros à conta. Escreva uma classe `SavingsAccountTester` que cria uma conta de poupança com um saldo inicial de US$ 1.000 e uma taxa de juros de 10%. Aplique então o método `addInterest` e imprima o saldo resultante. Também calcule o resultado esperado manualmente e o imprima.

★★ **Exercício P3.4.** Implemente uma classe `Employee`. Um funcionário tem um nome (uma string) e um salário (um `double`). Forneça um construtor com dois parâmetros:

```
public Employee(String employeeName, double currentSalary)
```

e métodos:

```
public String getName()
public double getSalary()
public void raiseSalary(double byPercent)
```

Esses métodos retornam o nome e o salário e aumentam o salário do funcionário de acordo com uma porcentagem. Exemplo de uso:

```
Employee harry = new Employee("Hacker, Harry", 50000);
harry.raiseSalary(10); // Harry ganha 10% de aumento
```

Forneça uma classe `EmployeeTester` que testa todos os métodos.

★★ **Exercício P3.5.** Implemente uma classe `Car` com as seguintes propriedades. Um carro tem um rendimento de combustível (medido em milhas/galão ou litros/km – escolha um) e uma quantidade de combustível no tanque de gasolina. O rendimento é especificado no construtor e o nível inicial de combustível é 0. Forneça um método `drive` que simula o uso do carro de acordo com certa distância, reduzindo a quantidade de gasolina no tanque de combustível. Também forneça os métodos `getGasInTank`, retornando a quantidade atual de gasolina no tanque de combustível, e `addGas`, para adicionar gasolina ao tanque de combustível. Exemplo de uso:

```
Car myHybrid = new Car(50); // 50 milhas por galão
myHybrid.addGas(20); // Tanque com 20 galões
myHybrid.drive(100); // Dirige por 100 milhas
double gasLeft = myHybrid.getGasInTank(); // Obtém a qtde de gasolina restante no tanque
```

Você pode supor que o método `drive` nunca é chamado com uma distância que consome mais gasolina do que a disponível. Forneça uma classe `CarTester` que testa todos os métodos.

★★ **Exercício P3.6.** Implemente uma classe `Student`. Para o propósito deste exercício, um estudante tem um nome e uma pontuação total no exame. Forneça um construtor apropriado e os métodos `getName()`, `addQuiz(int score)`, `getTotalScore()` e `getAverageScore()`. Para calcular o último, você também precisa armazenar o *número de exames* que o aluno fez.

Forneça uma classe `StudentTester` que testa todos os métodos.

★ **Exercício P3.7.** Implemente uma classe `Product`. Um produto tem um nome e um preço, por exemplo, `new Product("Toaster", 29.95)`. Forneça os métodos `getName`, `getPrice` e `reducePrice`. Forneça um programa `ProductPrinter` que cria dois produtos, imprime o nome e o preço, reduz os preços por US$ 5.00 e então imprime os preços novamente.

★★ **Exercício P3.8.** Forneça uma classe para escrever uma carta simples. No construtor, forneça os nomes do remetente e do destinatário:

```
public Letter(String from, String to)
```

Forneça um método

```
public void addLine(String line)
```

para adicionar uma linha de texto ao corpo da carta.

Forneça um método

```
public String getText()
```

que retorna o texto inteiro da carta. O texto tem a forma:

Prezado nome do destinatário:
linha em branco
primeira linha do corpo
segunda linha do corpo
. . .
última linha do corpo
linha em branco
Atenciosamente,
linha em branco
nome do remetente

Também forneça um programa `LetterPrinter` que imprime essa carta.

```
Dear John:

I am sorry we must part.
I wish you all the best.

Sincerely,

Mary
```

Construa um objeto da classe `Letter` e chame `addLine` duas vezes.

Dicas: (1) Utilize o método `concat` para formar uma string mais longa a partir de duas strings mais curtas. (2) A string especial `"\n"` representa uma nova linha. Por exemplo, a instrução

```
body = body.concat("Sincerely,").concat("\n");
```

adiciona uma linha contendo a string `"Sincerely,"` ao corpo da carta.

★★ **Exercício P3.9.** Escreva uma classe `Bug` que represente um inseto movendo-se sobre uma linha horizontal. O inseto se move para a direita ou para a esquerda. Inicialmente, ele se move para a direita, mas pode virar e mudar de direção. Em cada movimento, a posição muda de uma unidade na direção atual. Forneça um construtor

```
public Bug(int initialPosition)
```

e os métodos

```
public void turn()
public void move()
public int getPosition()
```

Exemplo de uso:

```
Bug bugsy = new Bug(10);
bugsy.move(); // agora a posição é 11
bugsy.turn();
bugsy.move(); // agora a posição é 10
```

Seu `BugTester` deve criar um inseto, fazer ele se mover, virar algumas vezes e imprimir a posição real e a esperada.

★★ **Exercício P3.10.** Implemente uma classe `Moth` que modele uma mariposa que voa ao longo de uma linha reta. A mariposa tem uma posição, a distância a partir de uma origem fixa. Quando a mariposa se move em direção a um ponto de luz, sua nova posição é a metade entre a posição antiga e a posição da fonte de luz. Forneça um construtor

```
public Moth(double initialPosition)
```

e os métodos

```
public void moveToLight(double lightPosition)
public void getPosition()
```

Seu `MothTester` deve criar uma mariposa, movê-la em direção a algumas fontes de luz e verificar se a posição da mariposa é a esperada.

★★ **Exercício P3.11.** Implemente uma classe `RoachPopulation` que simula o crescimento de uma população de baratas. O construtor recebe o tamanho da população inicial de baratas. O método `breed` simula o período em que as baratas procriam, o que dobra a sua população. O método `spray` simula a pulverização com um inseticida, o que reduz a população em 10%. O método `getRoaches` retorna o número atual de baratas. Um programa chamado `RoachSimulation` simula uma população que inicia com 10 baratas, procria, pulveriza e imprime a contagem de baratas. Repita o processo três vezes.

★★ **Exercício P3.12.** Implemente uma classe `VotingMachine` que pode ser utilizada para uma eleição simples. Crie os métodos para limpar o estado da máquina, votar em um democrata, votar em um republicano e obter a totalização dos votos para os dois partidos. Você ganha um crédito extra se seu programa totaliza a apuração dos votos do seu partido preferido depois das 20h da primeira terça-feira de novembro, mas funciona normalmente em todas as outras datas. (*Dica:* Utilize a classe `GregorianCalendar` – veja o Projeto de Programação 2.1.)

★★G **Exercício P3.13.** Desenhe um "alvo" – um conjunto de anéis concêntricos nas cores preto e branco alternadas. *Dica:* Preencha um círculo preto e depois um círculo branco menor por cima e assim por diante.

Seu programa deve ser composto das classes `BullsEye`, `BullsEyeComponent` e `BullsEyeViewer`.

★★G **Exercício P3.14.** Escreva um programa que desenha uma casa. Ela pode ser tão simples quanto a figura a seguir ou, se você preferir, mais elaborada (em 3D, um arranha-céu, com colunas de mármore no hall de entrada, qualquer coisa).

Implemente uma classe `House` e forneça um método `draw(Graphics2D g2)` que desenha a casa.

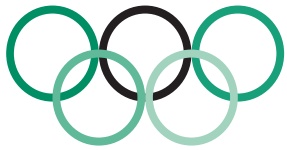
★★G **Exercício P3.15.** Estenda o Exercício P3.14 fornecendo um construtor `House` para especificar a posição e o tamanho da casa. Em seguida, preencha a tela com algumas casas de tamanhos diferentes.

★★G **Exercício P3.16.** Mude o programa visualizador de carros da Seção 3.9 para que os carros apareçam em cores diferentes. Cada objeto `Car` deve armazenar sua própria cor. Forneça as classes `Car` e `CarComponent` modificadas.

★★G **Exercício P3.17.** Altere a classe `Car` para que o tamanho de um carro possa ser especificado no construtor. Altere a classe `CarComponent` para que um dos carros tenha duas vezes o tamanho do exemplo original.

★★G **Exercício P3.18.** Escreva um programa para plotar a string "HELLO", utilizando apenas linhas e círculos. Não chame `drawString` e não utilize `System.out`. Crie as classes `LetterH`, `LetterE`, `LetterL` e `LetterO`.

★★G **Exercício P3.19.** Escreva um programa que exibe os anéis olímpicos. Aplique aos anéis as cores olímpicas.



Forneça uma classe `OlympicRingViewer` e uma classe `OlympicRingComponent`.

★★G **Exercício P3.20.** Crie um gráfico de barras para plotar o conjunto de dados a seguir. Rotule cada barra. Crie barras horizontais para facilitar a rotulação. Forneça uma classe `BarChartViewer` e uma classe `BarChartComponent`.

Nome da ponte	Maior vão livre (ft)
Golden Gate	4.200
Brooklyn	1.595
Delaware Memorial	2.150
Mackinac	3.800

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 3.1.** Neste projeto, você irá aprimorar a classe `BankAccount` e ver como a abstração e o encapsulamento permitem alterações evolutivas para os softwares.

Inicie com um aprimoramento simples: cobrando uma tarifa para cada depósito e saque. Forneça um mecanismo para configurar a tarifa e modifique os métodos `deposit` e `withdraw` de modo que a tarifa seja cobrada. Teste sua classe resultante e verifique se a tarifa é calculada corretamente.

Agora faça uma alteração mais complexa. O banco permitirá um número fixo de transações gratuitas (depósitos ou saques) todos os meses e cobrará pelas transações que excederem a cota gratuita. A taxa não é cobrada imediatamente, mas no fim do mês.

Forneça um novo método `deductMonthlyCharge` para a classe `BankAccount` que subtrai a taxa mensal e redefine a contagem de transações. Crie um programa de teste que verifica se as tarifas são calculadas corretamente ao longo de vários meses.

★★★ **Projeto 3.2.** Neste projeto, você irá explorar uma alternativa orientada a objetos para o programa "Hello, World" do Capítulo 1.

Inicie com uma classe `Greeter` simples que contém um único método, `sayHello`. Esse método deve *retornar* uma string sem imprimi-la. Use o BlueJ para criar dois objetos dessa classe e invoque seus métodos `sayHello`.

Isso é entediante – naturalmente, já que os dois objetos retornam a mesma resposta.

Aprimore a classe `Greeter` para que cada objeto produza uma saudação personalizada. Por exemplo, o objeto construído como `new Greeter("Dave")` deve dizer `"Hello, Dave"`. (Utilize o método `concat` para combinar strings a fim de formar uma string mais longa ou examine a Seção 4.6 para ver como você pode utilizar o operador `+` para o mesmo propósito.)

Adicione um método `sayGoodbye` à classe `Greeter`.

Por fim, adicione um método `refuseHelp` à classe `Greeter`. Ele deve retornar uma string como `"I am sorry, Dave. I am afraid I can't do that"`. ["Desculpe, Dave. Infelizmente, não posso fazer isso."]

Teste sua classe no BlueJ. Crie objetos que saúdam todo mundo em geral e Dave em particular, e invoque métodos a partir deles.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Os programadores que projetaram e implementaram a biblioteca Java.
2. Outros programadores que trabalham com o aplicativo de finanças pessoais.
3. `harrysChecking.withdraw(harrysChecking.getBalance())`
4. Adicione um parâmetro `accountNumber` aos construtores e um método `getAccountNumber`. Não há necessidade de um método `setAccountNumber` – o número da conta nunca muda depois da construção.

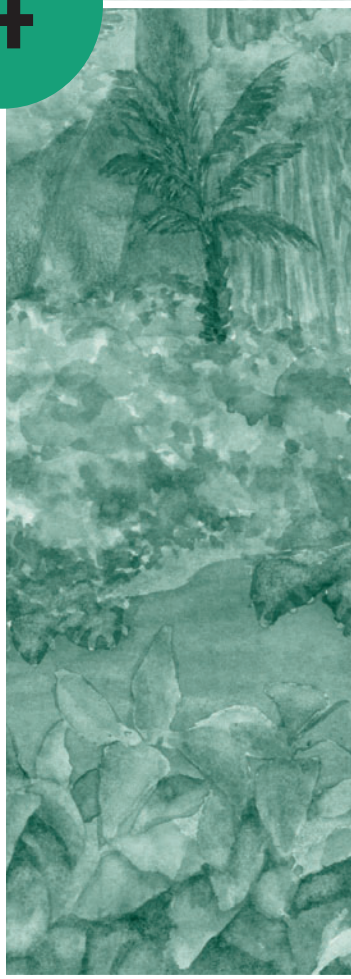
5.
```
/**
   Constrói uma nova conta bancária com um dado saldo inicial.
   @param accountNumber número dessa conta
   @param initialBalance saldo dessa conta
*/
```
6. A primeira frase da descrição do método deve descrever o método – ela é exibida isoladamente na tabela de resumo.
7. Um campo de instância

   `private int accountNumber;`

   precisa ser adicionado à classe.
8. Há quatro campos: `x`, `y`, `width`, `height`. Todos os campos são do tipo `int`.
9.
```
public int getWidth()
{
   return width;
}
```
10. Há mais de uma resposta correta. Uma possível implementação seria esta:
```
public void translate(int dx, int dy)
{
   int newx = x + dx;
   x = newx;
   int newy = y + dy;
   y = newy;
}
```
11. Um objeto `BankAccount`, nenhum objeto `BankAccountTester`. O propósito da classe `BankAccountTester` é apenas hospedar o método `main`.
12. Nesses ambientes, você pode emitir comandos interativos para construir objetos `BankAccount`, invocar métodos e exibir valores de retorno.
13. Variáveis de ambas pertencem a métodos – elas ganham vida quando o método é chamado e morrem quando o método é fechado. Elas diferem na maneira como são inicializadas. Variáveis de parâmetro são inicializadas com os valores de chamada; variáveis locais devem ser inicializadas explicitamente.
14. Um campo de instância, chamado `balance`. Três variáveis locais, uma chamada `harrysChecking` e duas chamadas `newBalance` (nos métodos `deposit` e `withdraw`); duas variáveis de parâmetro, as duas chamadas `amount` (nos métodos `deposit` e `withdraw`).
15. Um parâmetro implícito do tipo `BankAccount` chamado `this` e um parâmetro explícito do tipo `double` chamado `amount`.
16. Não é uma expressão válida. `this` é do tipo `BankAccount`; e a classe `BankAccount` não tem campo algum chamado `amount`.
17. Nenhum parâmetro implícito – o método é estático – e um parâmetro explícito, chamado `args`.
18. `CarComponent`
19. No método `draw` da classe `Car`, chame:
```
g2.fill(frontTire);
g2.fill(rearTire);
```
20. Dobre todas as medidas no método `draw` da classe `Car`.

Capítulo 4

Tipos de Dados Fundamentais



Objetivos do capítulo

- Entender números inteiros e de ponto flutuante
- Reconhecer as limitações dos tipos numéricos
- Tornar-se ciente das causas dos erros de overflow e erros de arredondamento
- Entender o uso adequado de constantes
- Escrever expressões aritméticas em Java
- Utilizar o tipo `String` para definir e manipular strings de caracteres
- Aprender a ler entrada de programa e produzir saída formatada

Este capítulo ensina a manipular números e cadeias de caracteres (strings) em Java. O objetivo deste capítulo é obter um entendimento sólido dos tipos de dados fundamentais em Java.

Você aprenderá as propriedades e as limitações dos tipos numéricos em Java e verá como manipular números e strings nos seus programas. Por fim, abrangeremos o importante tópico de entrada e saída, que permite implementar programas interativos.

Conteúdo do Capítulo

4.1 Tipos numéricos 146
Sintaxe 4.1: Coerção (Typecasting) 148
Tópico avançado 4.1: Números grandes ⊕
Tópico avançado 4.2: Números binários ⊕
Fato aleatório 4.1: O bug de ponto flutuante do Pentium ⊕

4.2 Constantes 149
Sintaxe 4.2: Definição de constante 151
Dica de qualidade 4.1: Não utilize números mágicos 154
Dica de qualidade 4.2: Escolha nomes descritivos para variáveis 154

4.3 Atribuição, incremento e decremento 155
Dica de produtividade 4.1: Evite um leiaute instável ⊕
Tópico avançado 4.3: Combinando atribuição e aritmética ⊕

4.4 Operações aritméticas e funções matemáticas 156
Erro comum 4.1: Divisão de inteiros 159
Erro comum 4.2: Parênteses desbalanceados 160
Dica de qualidade 4.3: Espaço em branco 160
Dica de qualidade 4.4: Fatorando código comum 161

4.5 Chamando métodos estáticos 161
Sintaxe 4.3: Chamada de método estático 162
Erro comum 4.3: Erros de arredondamento 163
Como fazer 4.1: Realizando cálculos 163

4.6 Strings 166
Dica de produtividade 4.2: Lendo relatórios de exceções 168
Tópico avançado 4.4: Seqüências de escape ⊕
Tópico avançado 4.5: Strings e o tipo char ⊕
Fato aleatório 4.2: Alfabetos internacionais ⊕

4.7 Lendo a entrada 169
Tópico avançado 4.6: Formatando números ⊕
Tópico avançado 4.7: Utilizando caixas de diálogo para entrada e saída ⊕

Em Java, um valor ou é uma referência a um objeto ou pertence a um dos oito *tipos primitivos* mostrados na Tabela 1.

Seis dos tipos primitivos são tipos numéricos, quatro deles para inteiros e dois para números de ponto flutuante.

Um cálculo numérico causa overflow se o resultado estiver fora do intervalo do tipo numérico.

Cada um dos tipos inteiros possui um intervalo diferente – o Tópico Avançado 4.2 explica por que os limites de intervalo estão relacionados a potências de dois. No geral, você utilizará o tipo `int` para quantidades inteiras. Mas, ocasionalmente, cálculos que envolvem inteiros podem causar *overflow*, ou "estouro". Isso acontece se o resultado de um cálculo exceder o intervalo do tipo numérico. Por exemplo:

```
int n = 1000000;
System.out.println(n * n);  // Imprime –727379968
```

O produto `n * n` é 10^{12}, o que é maior que o maior inteiro (aproximadamente $2 \cdot 10^9$). O resultado está truncado para caber em um `int`, produzindo um valor completamente errado. Infelizmente, não há nenhum aviso quando um estouro de inteiro acontece.

Se encontrar esse problema, a correção mais simples é utilizar o tipo `long`. O Tópico Avançado 4.1 mostra como utilizar o tipo `BigInteger` de precisão arbitrária na improvável situação de até mesmo o tipo `long` estourar.

Tabela 1 Tipos primitivos

Tipo	Descrição	Tamanho
`int`	Tipo inteiro, com intervalo –2.147.483.648. . . 2.147.483.647 (cerca de 2 bilhões)	4 bytes
`byte`	Tipo que descreve um único byte, com intervalo –128. . . 127	1 byte
`short`	Tipo inteiro curto, com intervalo –32768 . . . 32767	2 bytes
`long`	Tipo inteiro longo, com intervalo –9.223.372.036.854.775.808 . . . 9.223.372.036.854.775.807	8 bytes
`double`	Tipo ponto flutuante de dupla precisão, com um intervalo de aproximadamente $\pm 10^{308}$ e aproximadamente 15 dígitos decimais significativos	8 bytes
`float`	Tipo ponto flutuante de precisão simples, com um intervalo de aproximadamente $\pm 10^{38}$ e aproximadamente 7 dígitos decimais significativos	4 bytes
`char`	Tipo caractere, representando unidades de código no esquema de codificação Unicode (ver Tópico avançado 4.5)	2 bytes
`boolean`	Tipo com dois valores exclusivos, `false` ou `true` (ver Capítulo 5)	1 bit

O estouro normalmente não é um problema para números de ponto flutuante de dupla precisão. O tipo `double` tem um intervalo de aproximadamente $\pm 10^{308}$ e 15 dígitos significativos. Mas você quer evitar o tipo `float` – ele tem menos de 7 dígitos significativos. (Alguns programadores utilizam `float` para salvar em memória, se precisarem armazenar um conjunto muito grande de números sem muita precisão.)

Erros de arredondamento ocorrem quando uma conversão exata entre números não é possível.

Erros de arredondamento são uma questão mais séria com valores de ponto flutuante. Erros de arredondamento podem ocorrer quando você converte entre números binários e decimais ou entre números inteiros e números de ponto flutuante. Quando um valor não pode ser convertido exatamente, ele é arredondado para o número mais próximo. Pense neste exemplo:

```
double f = 4.35;
System.out.println(100 * f); // Imprime 434.99999999999994.
```

Esse problema é causado porque computadores representam números no sistema binário. No sistema de números binários, não há representação exata da fração 1/10, assim como não há representação exata da fração 1/3 = 0,33333 no sistema de números decimais. (Veja o Tópico Avançado 4.2 para informações adicionais.)

Por essa razão, o tipo `double` não é apropriado para cálculos financeiros. Neste livro, continuaremos a utilizar valores `double` para saldos bancários e outros valores monetários para que possamos manter nossos programas o mais simples possível. Entretanto, programas profissionais precisam utilizar os tipos `BigDecimal` para esse propósito – veja o Tópico Avançado 4.1.

Em Java, é válido atribuir um valor do tipo inteiro a uma variável de ponto flutuante:

```
int dollars = 100;
double balance = dollars; // OK
```

Mas a atribuição oposta é um erro: você não pode atribuir uma expressão de ponto flutuante a uma variável do tipo inteiro.

```
double balance = 13.75;
int dollars = balance; // Erro
```

Para superar esse problema, você pode converter o valor de ponto flutuante em um inteiro usando uma coerção (ou typecasting):

```
int dollars = (int) balance;
```

Utilize uma coerção (*nomeDoTipo*) para converter um valor em um tipo diferente.

A coerção `(int)` converte o valor de ponto flutuante `balance` em um inteiro descartando a parte fracionária. Por exemplo, se `balance` for 13.75, então `dollars` será configurado como 13.

A coerção instrui o compilador de que você concorda com a *perda de informações,* nesse caso, com a perda da parte fracionária. Você também pode aplicar uma coerção a outros tipos, como `(float)` ou `(byte)`.

Utilize o método `Math.round` para arredondar um número de ponto flutuante para o inteiro mais próximo.

Se quiser arredondar um número de ponto flutuante para o número inteiro mais próximo, utilize o método `Math.round`. Esse método retorna um inteiro `long`, pois números de ponto flutuante grandes não podem ser armazenados em um `int`.

```
long rounded = Math.round(balance);
```

Se `balance` for 13.75, então `rounded` será configurado como 14.

SINTAXE 4.1 Coerção (Typecasting)

(*nomeDoTipo*) *expressão*

Exemplo:

`(int) (balance * 100)`

Objetivo:

Converter uma expressão em um tipo diferente

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Quais são os tipos numéricos mais utilizados em Java?
2. Quando a coerção `(long) x` produz um resultado diferente a partir da chamada `Math.round(x)`?
3. Como você arredonda o valor `double x` para o valor `int` mais próximo, supondo que você saiba que ele é menor que $2 \cdot 10^9$?

TÓPICO AVANÇADO 4.1

Números grandes

O Tópico Avançado 4.1 mostra como utilizar os tipos `BigInteger` e `BigDecimal` para lidar com números realmente grandes ou para melhor controlar erros de arredondamento.

TÓPICO AVANÇADO 4.2

Números binários

O Tópico Avançado 4.2 discute como números são codificados no computador utilizando o sistema binário.

FATO ALEATÓRIO 4.1

O bug de ponto flutuante do Pentium

O Fato Aleatório 4.1 relata a história do bug de ponto flutuante do Intel Pentium, uma falha nesse processador amplamente vendido que causava multiplicações erradas em circunstâncias raras. Descoberto por um professor de matemática que utilizava um computador em uma pesquisa sobre números primos, esse bug resultou em um *recall* de todos os chips afetados.

4.2 Constantes

Em muitos programas, você precisa utilizar *constantes* numéricas – valores que não mudam e que são especialmente importantes para um determinado cálculo.

Um exemplo típico do uso de constantes é um cálculo que envolve valores de moedas, como este:

```
payment = dollars + quarters * 0.25 + dimes * 0.1
      + nickels * 0.05 + pennies * 0.01;
```

A maior parte do código está autodocumentada. Mas quatro quantidades numéricas, 0.25, 0.1, 0.05 e 0.01, são incluídas na expressão aritmética sem explicação alguma. Naturalmente, nesse caso, sabemos que o valor de um *nickel* é cinco centavos, o que explica os 0.05 e assim por diante. Mas se a próxima pessoa que precisar manter esse código residir em um outro país, talvez ela não saiba que um nickel é uma moeda de cinco centavos.

Portanto, uma boa idéia é utilizar nomes simbólicos para todos os valores, mesmo para aqueles que parecem óbvios. Eis uma versão mais clara do cálculo do total:

```
double quarterValue = 0.25;
double dimeValue = 0.1;
double nickelValue = 0.05;
double pennyValue = 0.01;
payment = dollars + quarters * quarterValue + dimes * dimeValue
      + nickels * nickelValue + pennies * pennyValue;
```

Uma variável final é uma constante. Depois de o valor ser configurado, ele não pode ser alterado.

Há um outro aprimoramento que podemos fazer. Há uma diferença entre as variáveis `nickels` e `nickelValue`. A variável `nickels` pode verdadeiramente variar ao longo da vida do programa, à medida que calculamos diferentes pagamentos. Mas `nickelValue` sempre é 0.05.

Em Java, constantes são identificadas com a palavra-chave `final`. Uma variável marcada como `final` nunca pode mudar depois de ser configurada. Se tentar mudar o valor de uma variável `final`, o compilador informará um erro e seu programa não compilará.

Utilize constantes identificadas para facilitar a leitura e manutenção dos seus programas.

Muitos programadores utilizam nomes com todas as letras maiúsculas para constantes (variáveis `final`), como `NICKEL_VALUE`. Dessa maneira, é fácil distinguir entre variáveis (com a maioria das letras em minúsculas) e constantes. Seguiremos essa convenção neste livro. Mas essa regra é uma questão de bom estilo, não um requisito da linguagem Java. O compilador não reclamará se você atribuir a uma variável `final` um nome com todas as letras em minúsculas.

Eis uma versão aprimorada do código que calcula o valor de um pagamento.

```
final double QUARTER_VALUE = 0.25;
final double DIME_VALUE = 0.1;
final double NICKEL_VALUE = 0.05;
final double PENNY_VALUE = 0.01;
payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
      + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

Freqüentemente, valores de constantes são necessários em vários métodos. Você deve então declará-los junto com os campos de instância de uma classe e marcá-los como `static` e `final`. Como dito, `final` indica que o valor é uma constante. A palavra-chave `static` significa que a constante pertence à classe – isso será explicado com mais detalhes no Capítulo 8.

```
public class CashRegister
{
   // Métodos
   . . .
   // Constantes
   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
   public static final double NICKEL_VALUE = 0.05;
   public static final double PENNY_VALUE = 0.01;
   // Campos de instância
   private double purchase;
   private double payment;
}
```

Declaramos as constantes como `public`. Não há perigo nisso porque constantes não podem ser modificadas. Métodos de outras classes podem acessar uma constante pública especificando primeiro o nome da classe em que ela é definida, em seguida um ponto e então o nome da constante, como `CashRegister.NICKEL_VALUE`.

A classe `Math` da biblioteca padrão define algumas constantes úteis:

```
public class Math
{
   . . .
   public static final double E = 2.7182818284590452354;
   public static final double PI = 3.14159265358979323846;
}
```

Você pode referenciar essas constantes como `Math.PI` e `Math.E` em qualquer um dos seus métodos. Por exemplo,

```
double circumference = Math.PI * diameter;
```

O programa de exemplo no final desta seção demonstra o uso de constantes. Esse programa mostra um refinamento da classe `CashRegister` do Quadro Como Fazer 3.1. A interface pública dessa classe foi modificada para solucionar um problema comum do negócio.

Caixas bancários sobrecarregados às vezes cometem erros na soma dos valores de moedas. Nossa classe `CashRegister` apresenta um método cujas entradas são *contagens de moedas.* Por exemplo, a chamada:

```
register.enterPayment(1, 2, 1, 1, 4);
```

insere um pagamento que consiste em um dólar, duas moedas de vinte e cinco centavos (*quarters*), uma moeda de dez centavos (*dime*), uma moeda de cinco centavos (*nickel*) e quatro moedas de um centavo (*pennies*). O método `enterPayment` calcula o valor total do pagamento, US$ 1.69. Como você pode ver a partir da listagem de código, o método usa constantes identificadas para os valores de moedas.

SINTAXE 4.2 Definição de constante

Em um método:

`final` *nomeDoTipo nomeDaVariável* `=` *expressão*`;`

Em uma classe:

especificadorDeAcesso `static final` *nomeDoTipo nomeDaVariável* `=` *expressão*`;`

Exemplo:

```
final double NICKEL_VALUE = 0.05;
public static final double LITERS_PER_GALLON = 3.785;
```

Objetivo:

Definir uma constante em um método ou em uma classe

ch04/cashregister/CashRegister.java

```
/**
   Uma caixa registradora soma as vendas e calcula o troco devido.
*/
public class CashRegister
{
   /**
      Constrói uma caixa registradora sem dinheiro.
   */
   public CashRegister()
   {
      purchase = 0;
      payment = 0;
   }

   /**
      Registra o preço de compra de um item.
      @param amount preço do item adquirido
   */
   public void recordPurchase(double amount)
   {
      purchase = purchase + amount;
   }

   /**
      Insere o pagamento recebido do cliente.
      @param dollars número de dólares no pagamento
      @param quarters número de moedas de vinte e cinco centavos no pagamento
      @param dimes número de moedas de dez centavos no pagamento
      @param nickels número de moedas de cinco centavos no pagamento
      @param pennies número de moedas de um centavo no pagamento
   */
   public void enterPayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   {
      payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
            + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
   }

   /**
      Calcula o troco devido e redefine a máquina para o próximo cliente.
      @return troco devido ao cliente
   */
   public double giveChange()
   {
      double change = payment - purchase;
      purchase = 0;
      payment = 0;
      return change;
   }

   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
```

```
53      public static final double NICKEL_VALUE = 0.05;
54      public static final double PENNY_VALUE = 0.01;
55
56      private double purchase;
57      private double payment;
58   }
```

ch04/cashregister/CashRegisterTester.java

```
1   /**
2      Essa classe testa a classe CashRegister
3   */
4   public class CashRegisterTester
5   {
6      public static void main(String[] args)
7      {
8         CashRegister register = new CashRegister();
9
10        register.recordPurchase(0.75);
11        register.recordPurchase(1.50);
12        register.enterPayment(2, 0, 5, 0, 0);
13        System.out.print("Change: ");
14        System.out.println(register.giveChange());
15        System.out.println("Expected: 0.25");
16
17        register.recordPurchase(2.25);
18        register.recordPurchase(19.25);
19        register.enterPayment(23, 2, 0, 0, 0);
20        System.out.print("Change: ");
21        System.out.println(register.giveChange());
22        System.out.println("Expected: 2.0");
23     }
24  }
```

Saída

```
Change: 0.25
Expected: 0.25
Change: 2.0
Espected: 2.0
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

4. Qual é a diferença entre as duas instruções a seguir?

   `final double CM_PER_INCH = 2.54;`

   e

   `public static final double CM_PER_INCH = 2.54;`

5. O que há de errado com a seguinte instrução?

   `double circumference = 3.14 * diameter;`

Dica de qualidade 4.1

Não utilize números mágicos

Um número mágico é uma constante numérica que aparece no seu código sem uma explicação. Por exemplo, considere o estranho exemplo a seguir que realmente ocorre na origem da biblioteca Java:

```
h = 31 * h + ch;
```

Por que 31? O número de dias em janeiro? Um a menos do que o número de bits em um inteiro? Na verdade, esse código calcula um "código de hash" a partir de uma string – um número que é derivado dos caracteres da string de uma maneira que diferentes strings quase certamente produzirão diferentes códigos de hash. O valor 31, na realidade, embaralha os valores dos caracteres de uma maneira elegante.

Uma solução melhor é utilizar uma constante identificada:

```
final int HASH_MULTIPLIER = 31;
h = HASH_MULTIPLIER * h + ch;
```

Você nunca deve utilizar números mágicos no seu código. Qualquer número que não seja completamente auto-explicativo deve ser declarado como uma constante identificada. Mesmo a constante cósmica mais razoável um belo dia irá mudar. Você acha que há 365 dias em um ano? Seus clientes em Marte ficarão muito magoados por causa desse ridículo preconceito. Crie uma constante

```
final int DAYS_PER_YEAR = 365;
```

A propósito, o artifício:

```
final int THREE_HUNDRED_AND_SIXTY_FIVE = 365;
```

é contraprodutivo e desaprovado.

Dica de qualidade 4.2

Escolha nomes descritivos para variáveis

Em álgebra, nomes de variáveis normalmente contêm só uma letra, como p ou A, talvez com um subscrito como p_1. Seria tentador poupar bastante digitação utilizando nomes curtos de variáveis nos seus programas Java:

```
payment = d + q * QV + di * DIV + n * NV + p * PV;
```

Compare isso com esta instrução:

```
payment = dollars + quarters * QUARTER_VALUE + dimes * DIME_VALUE
      + nickels * NICKEL_VALUE + pennies * PENNY_VALUE;
```

A vantagem é óbvia. Ler `dollars` é muito mais fácil do que ler `d` e então imaginar que ele significa "dólares".

Na programação prática, nomes de variáveis auto-explicativos são particularmente importantes quando programas são escritos por mais de uma pessoa. Pode ser óbvio para você que `d` significa dólares, mas isso será óbvio para a pessoa que irá atualizar seu código anos mais tarde, muito tempo depois que você foi promovido (ou deixou a empresa)? Da mesma forma, você conseguirá lembrar o que `d` significa quando examinar esse código daqui a seis meses?

4.3 Atribuição, incremento e decremento

O operador `=` é chamado de operador de atribuição. À esquerda, você precisa de um nome de variável. À direita pode ser um único valor ou uma expressão. O operador de atribuição configura a variável com o valor dado. Até aqui, isso é simples e direto. Mas agora veremos um uso mais interessante do operador de atribuição. Considere a instrução

```
items = items + 1;
```

Ela significa, "Calcule o valor da expressão `items + 1` e adicione o resultado novamente à variável `items`". (Veja Figura 1.) O efeito geral da execução dessa instrução é incrementar `items` de 1. Por exemplo, se `items` fosse 3 antes da execução da instrução, ele seria depois configurado como 4. (Essa instrução seria útil se a caixa registradora monitorasse o número de itens adquiridos.)

O sinal `=` *não* significa que o lado esquerdo é igual ao lado direito. Em vez disso, ele é uma instrução para copiar o valor à direita para a variável à esquerda. Não confunda essa operação de atribuição com o operador relacional = utilizado na álgebra e que indica igualdade. O operador de atribuição é uma instrução para fazer algo, a saber, atribuir um valor a uma variável. A igualdade matemática afirma que dois valores são iguais. Naturalmente, na matemática não faria sentido escrever que $i = i + 1$; nenhum inteiro pode ser igual a ele mesmo mais 1.

A atribuição para uma variável não é a mesma coisa que a igualdade matemática.

Os conceitos de atribuição e igualdade não têm relação entre si e é um pouco lamentável que a linguagem Java (seguindo o C e o C++) utilize `=` para denotar atribuição. Outras linguagens de programação utilizam um símbolo como `<-` ou `:=`, o que evita essa confusão.

Os operadores `++` e `--` incrementam e decrementam uma variável.

A operação de incremento é tão comum ao escrever programas que há uma abreviação especial para ela, a saber

```
items++;
```

Essa instrução também adiciona 1 a `items`. Mas é mais fácil de digitar e ler do que a instrução de atribuição explícita. Como você poderia supor, também há um operador de decremento `--`. A instrução

```
items--;
```

subtrai 1 de `items`.

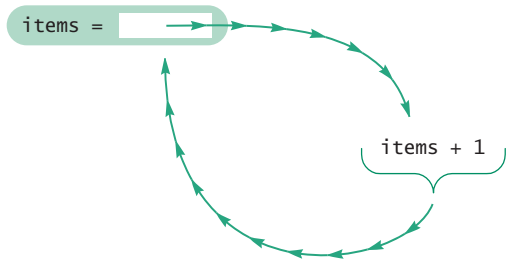


Figura 1
Incrementando uma variável.

Autoverificação da aprendizagem

6. Qual é o significado da instrução a seguir?

```
balance = balance + amount;
```

7. Qual é o valor de `n` após esta seqüência de instruções?

```
n--;
n++;
n--;
```

Dica de produtividade 4.1

Evite um leiaute instável

A Dica de Produtividade 4.1 sugere que você não gaste energia em embelezamentos que consomem tempo, como alinhar todos os operadores `=`.

Tópico avançado 4.3

Combinando atribuição e aritmética

O Tópico Avançado 4.3 abrange operadores especiais que combinam atribuição e aritmética. Por exemplo, `balance += amount` é equivalente a `balance = balance + amount`.

4.4 Operações aritméticas e funções matemáticas

Já vimos como adicionar, subtrair e multiplicar valores. A divisão é indicada por uma `/`, não uma barra de fração. Por exemplo,

$$\frac{a+b}{2}$$

torna-se

```
(a + b) / 2
```

Parênteses são utilizados da mesma maneira que na álgebra: para indicar em que ordem as subexpressões devem ser calculadas. Por exemplo, na expressão `(a + b) / 2`, a soma `a + b` é calculada primeiro e o resultado é dividido por 2. Em comparação, na expressão

```
a + b / 2
```

somente `b` é dividido por 2 e então a soma de `a` e `b / 2` é realizada. Assim como na notação algébrica regular, a multiplicação e a divisão têm precedência sobre a adição e a subtração. Por exemplo, na expressão `a + b / 2`, o `/` é executado primeiro, embora a operação `+` ocorra mais à esquerda.

Se os dois argumentos do operador / forem inteiros, o resultado será um inteiro e o resto será descartado.

A divisão funciona como você esperaria, contanto que pelo menos um dos números envolvidos seja um número de ponto flutuante. Isto é,

```
7.0 / 4.0
7 / 4.0
7.0 / 4
```

produzem 1.75. Mas se os dois números forem inteiros, o resultado da divisão será também um inteiro, com o resto descartado. Isto é,

```
7 / 4
```

é avaliado como 1, pois 7 dividido por 4 é 1 com um resto de 3 (que é descartado). Isso pode ser uma fonte sutil de erros de programação – veja o Erro comum 4.1.

O operador % calcula o resto de uma divisão.

Se você só estiver interessado no resto de uma divisão de inteiros, utilize o operador %:

```
7 % 4
```

é 3, o resto da divisão do inteiro 7 por 4. O símbolo % não tem análogo na álgebra. Ele foi escolhido porque é semelhante a / e a operação do cálculo do resto está relacionada com a divisão.

Eis um uso típico das operações / e % com inteiros. Suponha que você queira saber o troco que uma caixa registradora deve fornecer, utilizando valores separados para dólares e centavos. Você pode calcular esse valor como um inteiro, expresso em centavos, e então calcular o valor integral do dólar e o troco restante:

```
final int PENNIES_PER_NICKEL = 5;
final int PENNIES_PER_DIME = 10;
final int PENNIES_PER_QUARTER = 25;
final int PENNIES_PER_DOLLAR = 100;

// Calcula o valor total em moedas de um centavo
int total = dollars * PENNIES_PER_DOLLAR + quarters * PENNIES_PER_QUARTER
      + nickels * PENNIES_PER_NICKEL + dimes * PENNIES_PER_DIME + pennies;

// Usa a divisão de inteiro para converter em dólares, centavos
int dollars = total / PENNIES_PER_DOLLAR;
int cents = total % PENNIES_PER_DOLLAR;
```

Por exemplo, se `total` for 243, então `dollars` será configurado como 2 e `cents` como 43.

A classe `Math` contém os métodos `sqrt` e `pow` para calcular raízes quadradas e potências.

Para calcular x^n, escreva `Math.pow(x, n)`. Mas para calcular x^2 é infinitamente mais eficiente simplesmente calcular `x * x`.

Para obter a raiz quadrada de um número, utilize o método `Math.sqrt`. Por exemplo, $\sqrt{x}$ é escrito como `Math.sqrt(x)`.

Na álgebra, utilizamos frações, sobrescritos para expoentes e radicais para raízes quadradas a fim de organizar expressões em uma forma compacta bidimensional. Em Java, você precisa escrever todas as expressões de uma maneira linear. Por exemplo, a subexpressão

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

da fórmula quadrática se torna

```
(-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a)
```

A Figura 2 mostra como analisar essa expressão. Com expressões complexas como essas, nem sempre é fácil balancear os parênteses – veja Erro Comum 4.2.

A Tabela 2 mostra os métodos adicionais da classe `Math`. Entradas e saídas são números de ponto flutuante.

Tabela 2 Métodos matemáticos

Função	Retorna
`Math.sqrt(x)`	Raiz quadrada de x (≥ 0)
`Math.pow(x, y)`	x^y ($x > 0$ ou $x = 0$ e $y > 0$ ou $x < 0$ e y é um inteiro)
`Math.sin(x)`	Seno de x (x em radianos)
`Math.cos(x)`	Cosseno de x
`Math.tan(x)`	Tangente de x
`Math.asin(x)`	Arco seno ($\text{sen}^{-1}x \in [-\pi/2, \pi/2]$, $x \in [-1, 1]$)
`Math.acos(x)`	Arco cosseno ($\cos^{-1}x \in [0, \pi]$, $x \in [-1, 1]$)
`Math.atan(x)`	Arco tangente ($\text{tg}^{-1}x \in [-\pi/2, \pi/2]$)
`Math.atan2(y, x)`	Arco tangente ($\text{tg}^{-1}y/x \in [-\pi, \pi]$), x pode ser 0
`Math.toRadians(x)`	Converte x graus em radianos (isto é, retorna $x \cdot \pi/180$)
`Math.toDegrees(x)`	Converte x radianos em graus (isto é, retorna $x \cdot 180/\pi$)
`Math.exp(x)`	e^x
`Math.log(x)`	Logaritmo natural ($\ln(x)$, $x > 0$)
`Math.round(x)`	Inteiro mais próximo de x (como um `long`)
`Math.ceil(x)`	Menor inteiro $\geq x$ (como um `double`)
`Math.floor(x)`	Maior inteiro $\leq x$ (como um `double`)
`Math.abs(x)`	Valor absoluto $\lvert x \rvert$
`Math.max(x, y)`	O maior de x e y
`Math.min(x, y)`	O menor de x e y

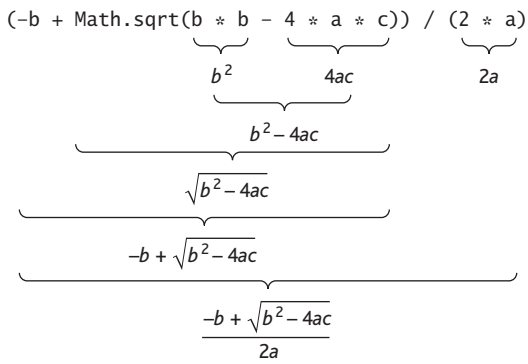


Figura 2 Analisando uma expressão.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

8. Qual é o valor de `1729 / 100`? De `1729 % 100`?
9. Por que a instrução a seguir não calcula a média de `s1`, `s2` e `s3`?

   `double average = s1 + s2 + s3 / 3; // Erro`
10. Qual é o valor de `Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))` em notação matemática?

ERRO COMUM 4.1

Divisão de inteiros

É lamentável que Java utilize o mesmo símbolo, /, tanto para divisão de inteiros como para divisão de números de ponto flutuante. Estas são operações bem diferentes. É um erro comum utilizar a divisão de inteiro acidentalmente. Considere este segmento do programa que calcula a média de três inteiros.

```
int s1 = 5; // Pontuação no teste 1
int s2 = 6; // Pontuação no teste 2
int s3 = 3; // Pontuação no teste 3
double average = (s1 + s2 + s3) / 3;  // Erro
System.out.print("Your average score is ");
System.out.println(average);
```

O que poderia estar errado nisso? Naturalmente, a média de `s1`, `s2` e `s3` é:

$$\frac{s_1 + s_2 + s_3}{3}$$

Aqui, entretanto, o / não significa divisão no sentido matemático. Ele indica a divisão de inteiros, porque os valores `s1 + s2 + s3` e 3 são todos inteiros. Por exemplo, se a pontuação for 14, a média será calculada como 4, o resultado da divisão do inteiro 14 por 3.

Esse inteiro 4 é então adicionado à variável de ponto flutuante `average`. O correto é transformar um numerador ou um denominador em um número de ponto flutuante:

```
double total = s1 + s2 + s3;
double average = total / 3;
```

ou

```
double average = (s1 + s2 + s3) / 3.0;
```

Erro comum 4.2

Parênteses desbalanceados

Considere a expressão

```
1.5 * ((-(b - Math.sqrt(b * b - 4 * a * c)) / (2 * a))
```

O que há de errado com ela? Conte os parênteses. Há cinco parênteses de abertura (e quatro parênteses de fechamento). Os parênteses não estão balanceados. Esse tipo de erro de digitação é muito comum em expressões complexas. Agora considere esta expressão.

```
1.5 * (Math.sqrt(b * b - 4 * a * c))) - ((b / (2 * a))
```

Essa expressão tem cinco parênteses de abertura (e cinco parênteses de fechamento), mas ainda não está correta. No meio da expressão,

```
1.5 * (Math.sqrt(b * b - 4 * a * c))) - ((b / (2 * a))
```

há apenas dois parênteses de abertura (mas três parênteses de fechamento), o que é um erro. No meio de uma expressão, a contagem dos parênteses de abertura deve ser maior que ou igual à contagem dos parênteses de fechamento e, no fim da expressão, as duas contagens devem ser idênticas.

Eis um truque simples para facilitar a contagem sem o uso de lápis e papel. É difícil para o cérebro manter duas contagens simultaneamente, portanto, mantenha apenas uma contagem ao analisar a expressão. Inicie com 1 no primeiro parêntese de abertura; adicione 1 sempre que você vir um parêntese de abertura; subtraia 1 sempre que vir um parêntese de fechamento. Fale os números em voz alta à medida que analisa a expressão. Se a contagem cair abaixo de zero ou se não for zero no fim, os parênteses não estão balanceados. Por exemplo, ao analisar a expressão anterior, você murmuraria:

```
1.5 * (Math.sqrt(b * b - 4 * a * c) )  ) - ((b / (2 * a))
      1         2                  1 0 -1
```

e encontraria o erro.

Dica de qualidade 4.3

Espaço em branco

Para o compilador não é importante se você escreve todo o seu programa em uma única linha ou coloca cada símbolo em uma linha separada. Para o leitor humano, porém, isso é muito importante. Você deve utilizar linhas em branco para agrupar visualmente seu código em seções. Por exemplo, você pode sinalizar ao leitor que um prompt de saída e a instrução de entrada correspondente estão relacionados inserindo uma linha

em branco antes e depois do grupo. Você encontrará muitos exemplos nas listagens de código-fonte neste livro.

O espaço em branco dentro de expressões também é importante. É mais fácil de ler

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

do que

```
x1=(-b+Math.sqrt(b*b-4*a*c))/(2*a);
```

Simplesmente coloque espaços em torno de todos os operadores `+ - * / % =`. Mas não coloque um espaço depois de um menos unário: um `-` utilizado para negar uma única quantidade, como em `-b`. Dessa maneira, ele pode ser facilmente distinguido de um menos binário, como em `a - b`. Não coloque espaços entre um nome de método e os parênteses, mas adicione um espaço depois de cada palavra-chave de Java. Isso facilita ver se o `sqrt` em `Math.sqrt(x)` é um nome de método, enquanto o `if` em `if (x > 0) . . .` é uma palavra-chave.

DICA DE QUALIDADE 4.4

Fatorando código comum

Suponha que você queira localizar as duas soluções da equação quadrática $ax^2 + bx + c = 0$. A fórmula quadrática nos diz que as soluções são:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Em Java, não há análogo à operação ±, que indica como obter duas soluções simultaneamente. As duas soluções devem ser calculadas separadamente:

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
x2 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

Essa abordagem tem dois problemas. Primeiro, o cálculo de `Math.sqrt(b * b - 4 * a * c)` é executado duas vezes, o que é uma perda de tempo. Segundo, sempre que o mesmo código é replicado, aumenta a possibilidade de ocorrer um erro de digitação. O melhor é fatorar o código comum:

```
double root = Math.sqrt(b * b - 4 * a * c);
x1 = (-b + root) / (2 * a);
x2 = (-b - root) / (2 * a);
```

Você poderia ir ainda mais longe e fatorar o cálculo de `2 * a`, mas o ganho no uso da fatoração em cálculos muito simples é muito pequeno para justificar o esforço.

4.5 Chamando métodos estáticos

Na seção anterior, vimos a classe `Math`, que contém uma coleção de métodos úteis para executar cálculos matemáticos. Esses métodos têm uma forma especial: eles são métodos estáticos, que não operam em um objeto.

Isto é, você não chama

```
double x = 4;
double root = x.sqrt(); // Erro
```

porque, em Java, números não são objetos, portanto você nunca invoca um método em um número. Em vez disso, você passa um número como um parâmetro explícito para um método, colocando o número entre parênteses depois do nome do método. Por exemplo, o valor do número `x` pode ser um parâmetro do método `Math.sqrt`: `Math.sqrt(x)`.

Um método estático não opera em um objeto.

Essa chamada faz parecer que o método `sqrt` é aplicado a um objeto chamado `Math`, porque `Math` vem antes de `sqrt` assim como `harrysChecking` vem antes de `getBalance` em uma chamada de método `harrysChecking.getBalance()`. Mas `Math` é uma classe, não um objeto. Um método como `Math.round` que não opera em objeto algum é chamado de método *estático*. (O termo "estático" é um remanescente histórico das linguagens de programação C e C++. Ele não tem nada a ver com o significado normal dessa palavra.) Métodos estáticos não operam em objetos, mas são definidos dentro das classes. Você deve especificar a classe a que o método `sqrt` pertence – por isso é chamada de `Math.sqrt(x)`.

Como você pode dizer se `Math` é uma classe ou um objeto? Todas as classes na biblioteca Java iniciam com uma letra maiúscula (como `System`). Objetos e métodos iniciam com uma letra minúscula (como `out` e `println`). (Você pode diferenciar entre objetos e métodos porque as chamadas de método são seguidas por parênteses.) Portanto, `System.out.println()` indica uma chamada do método `println` no objeto `out` dentro da classe `System`. Por outro lado, `Math.sqrt(x)` indica uma chamada ao método `sqrt` da classe `Math`. Esse uso de letras maiúsculas e minúsculas é meramente uma convenção, não uma regra da linguagem Java. É, porém, uma convenção que os autores das bibliotecas de classes Java devem seguir consistentemente. Você deve fazer o mesmo nos seus programas. Se atribuir nomes a objetos ou métodos que iniciam com letras maiúsculas, provavelmente irá confundir outros programadores que trabalham com você. Portanto, recomendamos veementemente que siga a convenção para atribuição de nomes padrão.

SINTAXE 4.3 Chamada de método estático

NomeDaClasse.nomeDoMétodo(parâmetros)

Exemplo:

`Math.sqrt(4)`

Objetivo:

Invocar um método estático (um método que não opera em um objeto) e fornecer seus parâmetros

AUTOVERIFICAÇÃO DA APRENDIZAGEM

11. Por que você não pode chamar `x.pow(y)` para calcular x^y?
12. A chamada `System.out.println(4)` é uma chamada de método estático?

ERRO COMUM 4.3

Erros de arredondamento

Erros de arredondamento são uma realidade quando calculamos números de ponto flutuante. Provavelmente você mesmo já tenha encontrado esse fenômeno em cálculos manuais. Se calcular 1/3 para duas casas decimais, você obterá 0.33. Multiplicando novamente por 3, você obtém 0.99, e não 1.00.

No hardware do processador, números são representados no sistema de números binários, não em decimais. Erros de arredondamento continuam a aparecer quando os dígitos binários são perdidos. Eles simplesmente aparecem em lugares diferentes do que aqueles que você poderia esperar. Eis um exemplo:

```
double f = 4.35;
int n = (int) (100 * f);
System.out.println(n); // Imprime 434!
```

Naturalmente, cem vezes 4.35 dá 435, mas o programa imprime 434.

Computadores representam números no sistema binário (ver Tópico Avançado 4.2). No sistema binário, não há representação exata para 4.35, assim como não há representação exata para 1/3 no sistema decimal. A representação utilizada pelo computador é um pouco menor que 4.35, assim 100 multiplicado por esse valor é um pouco menor que 435. Quando um valor de ponto flutuante é convertido em um inteiro, toda a parte fracionária é descartada, mesmo se ela for quase 1. Como resultado, o inteiro 434 é armazenado em `n`. Correção: utilize `Math.round` para converter números de ponto flutuante em inteiros. O método `round` retorna o inteiro *mais próximo*.

```
int n = (int) Math.round(100 * f);  // OK, n é 435
```

COMO FAZER 4.1

Realizando cálculos

Muitos problemas de programação requerem o uso de fórmulas matemáticas para calcular valores. Nem sempre é óbvio transformar o enunciado de um problema em uma seqüência de fórmulas matemáticas e, em última instância, em instruções na linguagem de programação Java.

Passo 1 Entenda o problema: Quais são as entradas? Quais são as saídas desejadas?

Por exemplo, suponha que sua tarefa seja simular uma máquina de venda de selos de correio. Um cliente insere dinheiro na máquina de venda. Então o cliente pressiona um botão "First class stamps" (selos para cartas via aérea). Em troca, a máquina automática libera o número de selos para cartas via aérea pelo qual o cliente pagou. (Um selo de primeira classe custava 39 centavos de dólar nos Estados Unidos na época em que este livro foi escrito.) Por fim, o cliente pressiona o botão "Penny stamps" (selos de 1 centavo) e a máquina fornece o troco em selos de um centavo.

Nesse problema, há uma entrada:

- O valor que o cliente insere

Há duas saídas desejadas:

- O número de selos para cartas via aérea que a máquina retorna
- O número de selos de um centavo que a máquina retorna

Passo 2 Formule os exemplos manualmente.

Esse é um passo muito importante. Se você não puder calcular algumas soluções manualmente, é provável que não consiga escrever um programa que automatize o cálculo.

Suponha que um selo de primeira classe custe 39 centavos e o cliente insira US$ 1.00. Isso é suficiente para dois selos (78 centavos), mas não é suficientemente para três selos (US$ 1.17). Portanto, a máquina retorna dois selos para cartas via aérea e 22 selos de um centavo.

Passo 3 Encontre equações matemáticas que calculem as respostas.

Dada uma quantia de dinheiro e fornecido o preço de um selo para cartas via aérea, como você pode calcular quantos selos para cartas via aérea podem ser comprados com esse dinheiro? Fica claro que a resposta está relacionada ao quociente:

$$\frac{\text{quantia de dinheiro}}{\text{preço do selo para cartas via aérea}}$$

Por exemplo, suponha que o cliente pague US$ 1.00. Utilize uma calculadora de bolso para calcular o quociente: $1.00/$0.39 ≈ 2.5641.

Como você obtém "2 selos" a partir de 2.5641? É a parte inteira. Descartando a parte fracionária, você obtém o número total de selos adquirido pelo cliente.
Em notação matemática,

$$\text{quantidade de selos para cartas via aérea} = \left\lfloor \frac{\text{dinheiro}}{\text{preço do selo para cartas via aérea}} \right\rfloor$$

onde $\lfloor x \rfloor$ indica o maior inteiro $\leq x$. Essa função às vezes é chamada de função de base ("floor function").

Agora você sabe como calcular o número de selos fornecidos quando o cliente pressiona o botão "First class stamps". Quando o cliente recebe os selos, a quantia de dinheiro é reduzida de acordo com o valor dos selos adquiridos. Por exemplo, se o cliente receber dois selos, o dinheiro restante é US$ 0.22 – a diferença entre $1.00 e 2 · $0.39. Eis a fórmula geral:

$$\text{dinheiro que sobrou} = \text{dinheiro} - \text{número de selos para cartas via aérea} \cdot \text{preço do selo para cartas via aérea}$$

Quantos selos de um centavo o dinheiro que sobrou compra? Isso é fácil. Se $0.22 é o que ficou, o cliente obtém 22 selos. Em geral, o número de selos de um centavo

$$\text{número de selos de um centavo} = 100 \cdot \text{dinheiro que sobrou}$$

Passo 4 Transforme as equações matemáticas em instruções Java.

Em Java, você pode calcular a parte inteira de um valor de ponto flutuante não-negativo aplicando uma coerção `(int)`. Portanto, você pode calcular o número de selos de primeira classe com a instrução a seguir:

```
firstClassStamps = (int) (money / FIRST_CLASS_STAMP_PRICE);
money = money - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
```

Por fim, o número de selos de um centavo é

```
pennyStamps = 100 * money;
```

Isso, porém, não está muito certo. O valor de `pennyStamps` deve ser um inteiro, mas o número do lado direito é um número de ponto flutuante. Portanto, a instrução correta é:

```
pennyStamps = (int) Math.round(100 * money);
```

Passo 5 Construa uma classe que execute seus cálculos.

O Como fazer 3.1 explica como desenvolver uma classe localizando métodos e variáveis de instância. No nosso caso, podemos encontrar três métodos:

- `void insert(double amount)`
- `int giveFirstClassStamps()`
- `int givePennyStamps()`

O estado de uma máquina de venda automática pode ser descrito pela quantidade de dinheiro que o cliente tem disponível para compras. Portanto, fornecemos uma variável de instância, `money`.

Eis a implementação:

```
public class StampMachine
{
   public StampMachine()
   {
      money = 0;
   }
   public void insert(double amount)
   {
      money = money + amount;
   }
   public int giveFirstClassStamps()
   {
      int firstClassStamps = (int) (money / FIRST_CLASS_STAMP_PRICE);
      money = money - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
      return firstClassStamps;
   }
   public int givePennyStamps()
   {
      int pennyStamps = (int) Math.round(100 * money);
      money = 0;
      return pennyStamps;
   }
   public static final double FIRST_CLASS_STAMP_PRICE = 0.39;
   private double money;
}
```

Passo 6 Teste sua classe.

Execute um programa de teste (ou utilize um ambiente integrado, como o BlueJ) para verificar se os valores que sua classe calcula são os mesmos valores que você calculou manualmente. No nosso exemplo, tente as instruções

```
StampMachine machine = new StampMachine();
machine.insert(1);
System.out.print("First class stamps: ");
System.out.println(machine.giveFirstClassStamps());
System.out.println("Expected: 2");
System.out.print("Penny stamps: ");
System.out.println(machine.givePennyStamps());
System.out.println("Expected: 22);
```

Verifique se o resultado é

```
First class stamps: 2
Expected: 2
Penny stamps: 22
Expected: 22
```

4.6 Strings

Depois dos números, strings são o tipo de dados mais importante que a maioria dos programas usa. Uma string é uma seqüência de caracteres, como `"Hello, World!"`. Em Java, strings ficam entre aspas, que não são parte da string. Observe que, diferentemente dos números, strings são objetos. (Você pode dizer que `String` é o nome de uma classe porque ela inicia com uma letra maiúscula. Os tipos primitivos `int` e `double` iniciam com letras minúsculas.)

Uma string é uma seqüência de caracteres. Strings são objetos da classe `String`.

O número de caracteres em uma string é chamado de comprimento da string. Por exemplo, o comprimento de `"Hello, World!"` é 13. Você pode calcular o comprimento de uma string com o método `length`.

```
int n = message.length();
```

Uma string de comprimento zero, que não contém nenhum caractere, é chamada string vazia e é escrita como `""`.

Utilize o operador + para agrupar strings e formar uma string mais longa.

```
String name = "Dave";
String message = "Hello, " + name;
```

Strings podem ser concatenadas, isto é, ligadas uma a outra para produzir uma string nova e mais longa. A concatenação de strings é indicada pelo operador +.

O operador + concatena duas strings, desde que um dos argumentos, à esquerda ou à direita de um operador +, seja uma string. O outro argumento é automaticamente forçado a também se tornar uma string para que ambas sejam concatenadas.

Por exemplo, considere este código:

```
String a = "Agent";
int n = 7;
String bond = a + n;
```

Sempre que um dos argumentos do operador + for uma string, o outro argumento é convertido em uma string.

Como `a` é uma string, `n` é convertido do inteiro 7 para a string `"7"`. As duas strings `"Agent"` e `"7"` são então concatenadas para formar a string `"Agent7"`.

Essa concatenação é muito útil para reduzir o número de instruções `System.out.print`. Por exemplo, você pode combinar:

```
System.out.print("The total is ");
System.out.println(total);
```

com a chamada

```
System.out.println("The total is " + total);
```

A concatenação `"The total is " + total` calcula uma única string que consiste na string `"The total is "`, seguida pela string equivalente ao número `total`.

Pode ocorrer que uma string contenha um número, normalmente inserido pelo usuário. Por exemplo, suponha que a variável string `input` contenha o número `"19"`. Para obter o valor inteiro `19`, utilize o método `parseInt` estático da classe `Integer`.

```
int count = Integer.parseInt(input);
   // count é o inteiro 19
```

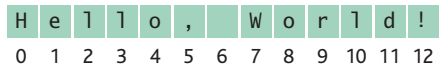


Figura 3 Posições da string.

Se uma string contiver os dígitos de um número, você pode utilizar o método `Integer.parseInt` ou o `Double.parseDouble` para obter o valor do número.

Para converter uma string que contém dígitos de ponto flutuante no valor de ponto flutuante, utilize o método estático `parseDouble` da classe `Double`. Por exemplo, suponha que `input` seja a string `"3.95"`.

```
double price = Double.parseDouble(input);
   // price é o número de ponto flutuante 3.95
```

Mas se a string contiver espaços ou outros caracteres que não podem ocorrer em números, ocorrerá um erro. Por enquanto, iremos supor que a entrada do usuário nunca contém caracteres inválidos.

Utilize o método `substring` para extrair uma parte de uma string.

O método `substring` calcula substrings de uma string. A chamada

```
s.substring(start, pastEnd)
```

retorna uma string composta dos caracteres na string `s`, iniciando na posição `start` e contendo todos os caracteres até, mas sem incluir, a posição `pastEnd`. Eis um exemplo:

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub é "Hello"
```

Posições em strings são contadas iniciando em 0.

A operação `substring` cria uma string que consiste em cinco caracteres pegos da string `greeting`. Um aspecto curioso da operação `substring` é a numeração das posições inicial e final. A primeira posição na string é rotulada como 0; a segunda, como 1; e assim por diante. Por exemplo, a Figura 3 mostra os números de posição na string `greeting`.

A posição do último caractere (12 para a string `"Hello, World!"`) é sempre o comprimento da string menos um.

Vamos pensar como extrair a substring `"World"`. Conte os caracteres que iniciam em 0, não em 1. Você descobre que `W`, o oitavo caractere, tem número de posição 7. O primeiro caractere que você não quer, `!`, é o caractere na posição 12 (veja Figura 4). Portanto, o comando apropriado de substring é

```
String sub2 = greeting.substring(7, 12);
```

É curioso que você precise especificar a posição do primeiro caractere que você quer e então a do primeiro caractere que você não quer. Há uma vantagem nessa configuração. Você pode calcular facilmente o comprimento da substring: Ele é `pastEnd - start`. Por exemplo, a string `"World"` tem comprimento 12 – 7 = 5.

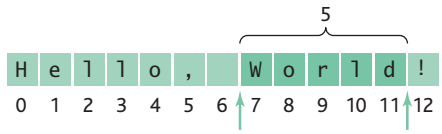


Figura 4 Extraindo uma substring.

Se você omitir o segundo parâmetro do método `substring`, então todos os caracteres a partir da posição inicial até o fim da string são copiados. Por exemplo,

```
String tail = greeting.substring(7); // Copia todos os caracteres da posição 7 em
//diante
```

configura `tail` como a string `"World!"`.

Se fornecer uma posição inválida para a string (um número negativo ou um valor maior que o comprimento da string), seu programa termina com uma mensagem de erro.

Nesta seção, fizemos a suposição de que cada caractere em uma string ocupa uma única posição. Infelizmente, essa suposição não é bem correta. Se você processar strings que contêm caracteres de outros alfabetos internacionais ou símbolos especiais, alguns desses caracteres podem ocupar duas posições – veja o Tópico Avançado 4.5.

Autoverificação da aprendizagem

13. Supondo que a variável `String s` contenha o valor `"Agent"`, qual é o efeito da atribuição `s = s + s.length()`?
14. Supondo que a variável `String river` contenha o valor `"Mississippi"`, qual é o valor de `river.substring(1, 2)`? E de `river.substring(2, river.length() - 3)`?

Dica de produtividade 4.2

Lendo relatórios de exceções

Freqüentemente haverá programas que terminam e exibem uma mensagem de erro, como

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
      String index out of range: -4
   at java.lang.String.substring(String.java:1444)
   at Homework1.main(Homework1.java:16)
```

Um número surpreendente de estudantes simplesmente desiste nesse ponto, dizendo "não funcionou", ou "meu programa deu pau", sem nunca ler a mensagem de erro. De fato, o formato do relatório de exceção não é muito amigável. Mas é fácil decifrá-lo.

Depois de examinar mais minuciosamente a mensagem de erro, você observará duas informações úteis:

1. O nome da exceção, como `StringIndexOutOfBoundsException`
2. O número da linha do código que continha a instrução que causou a exceção, como `Homework1.java:16`

O nome da exceção sempre está na primeira linha do relatório e termina em `Exception`. Se você recebe uma `StringIndexOutOfBoundsException`, há um problema de acesso a uma posição inválida em uma string. Essa é uma informação útil.

O número da linha do código com problemas é um pouco mais difícil de determinar. O relatório de exceção contém o rastreamento da pilha inteira – isto é, o nome de todos os métodos pendentes quando aconteceu a exceção. A primeira linha do rastreamento

da pilha é o método que realmente gerou a exceção. A última linha do rastreamento de pilha é uma linha em `main`. Na maioria das vezes essa exceção foi lançada por um método que está na biblioteca padrão. Procure a primeira linha no seu código que aparece no relatório de exceção. Por exemplo, pule a linha que se refere a

```
java.lang.String.substring(String.java:1444)
```

A próxima linha no nosso exemplo menciona um número de linha no seu código, `Homework1.java`. Depois de descobrir o número da linha no seu código, abra o arquivo, vá até essa linha e a examine! Na grande maioria dos casos, saber o nome da exceção e a linha que a causou torna completamente óbvio descobrir o que deu errado para que você possa corrigir o erro facilmente.

TÓPICO AVANÇADO 4.4

Seqüências de escape

O Tópico Avançado 4.4 mostra como você pode inserir caracteres especiais (como aspas ou quebras de linha) em strings.

TÓPICO AVANÇADO 4.5

Strings e o tipo char

O Tópico Avançado 4.5 discute o tipo `char`. Strings são compostas de unidades de código do tipo `char`. Para a maioria das tarefas de programação, simplesmente utilize strings de comprimento 1 em vez de valores `char`.

FATO ALEATÓRIO 4.2

Alfabetos internacionais

O Fato Aleatório 4.2 explica como o conjunto de caracteres Unicode fornece uma codificação para todos os caracteres que estão em uso no mundo, incluindo caracteres acentuados, alfabetos como o hebraico ou tailandês e milhares de ideogramas utilizados na China, no Japão e na Coréia.

4.7 Lendo a entrada

Utilize a classe `Scanner` para ler a entrada de teclado em uma janela de console.

Os programas Java que você criou até agora construíam objetos, chamavam métodos, imprimiam resultados e fechavam. Eles não eram interativos e não recebiam entrada de usuário. Nesta seção, você aprenderá um método para ler a entrada fornecida pelo usuário.

Como a saída é enviada a `System.out`, você poderia pensar que `System.in` é utilizado para a entrada. Infelizmente, não é tão simples assim. Quando Java foi inicialmente projetado, não foi dada muita atenção à leitura da entrada de teclado. Todo mundo pensou que os programadores produziriam interfaces gráficas com o usuário utilizando campos de texto e menus. `System.in` recebeu um conjunto mínimo de recursos – ele só pode ler um byte por vez. Finalmente, na versão Java 5, uma classe `Scanner` foi adicionada que permite ler entrada de teclado de uma maneira conveniente.

Para construir um objeto `Scanner`, simplesmente passe o objeto `System.in` para o construtor `Scanner`:

```
Scanner in = new Scanner(System.in);
```

Você pode criar um scanner a partir de um fluxo de entrada qualquer (como um arquivo), mas, em geral, você vai querer utilizar um scanner para ler a entrada de teclado a partir de `System.in`.

Com um scanner você pode utilizar os métodos `nextInt` ou `nextDouble` para ler o próximo inteiro ou próximo número de ponto flutuante.

```
System.out.print("Enter quantity: ");
int quantity = in.nextInt();

System.out.print("Enter price: ");
double price = in.nextDouble();
```

Quando o método `nextInt` ou `nextDouble` é chamado, o programa espera até que o usuário digite um número e pressione a tecla Enter. Você sempre deve fornecer instruções para o usuário (como `"Enter quantity:"`) antes de chamar um método `Scanner`. Essa instrução é chamada *prompt*.

O método `nextLine` retorna a próxima linha de entrada (até que o usuário pressione a tecla Enter) como um objeto `String`. O método `next` retorna a próxima *palavra,* terminada por um *espaço em branco,* isto é, um espaço, no fim de uma linha ou uma tabulação.

```
System.out.print("Enter city: ");
String city = in.nextLine();

System.out.print("Enter state code: ");
String state = in.next();
```

Aqui, utilizamos o método `nextLine` para ler o nome de uma cidade que pode consistir em múltiplas palavras, como `San Francisco`. Utilizamos o método `next` para ler o código do estado (como `CA`), que consiste em uma única palavra.

Eis um exemplo de uma classe que recebe a entrada de usuário. Essa classe usa a classe `CashRegister` e simula uma transação em que um usuário compra um item, paga e recebe troco.

Chamamos essa classe `CashRegisterSimulator`, não `CashRegisterTester`. Reservamos o sufixo `Tester` para classes cujo único propósito é testar outras classes.

ch04/cashregister/CashRegisterSimulator.java

```
1  import java.util.Scanner;
2
3  /**
4     Esse programa simula uma transação em que um usuário paga por um item
5     e recebe troco.
6  */
7  public class CashRegisterSimulator
8  {
9    public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12
13       CashRegister register = new CashRegister();
14
15       System.out.print("Enter price: ");
16       double price = in.nextDouble();
17       register.recordPurchase(price);
18
19       System.out.print("Enter dollars: ");
20       int dollars = in.nextInt();
21       System.out.print("Enter quarters: ");
22       int quarters = in.nextInt();
23       System.out.print("Enter dimes: ");
24       int dimes = in.nextInt();
25       System.out.print("Enter nickels: ");
26       int nickels = in.nextInt();
27       System.out.print("Enter pennies: ");
28       int pennies = in.nextInt();
29       register.enterPayment(dollars, quarters, dimes, nickels, pennies);
30
31       System.out.print("Your change: ");
32       System.out.println(register.giveChange());
33    }
34 }
```

Saída

```
Enter price: 7.55
Enter dollars: 10
Enter quarters: 2
Enter dimes: 1
Enter nickels: 0
Enter pennies: 0
Your change: 3.05
```

Autoverificação da aprendizagem

15. Por que a entrada não pode ser lida diretamente de `System.in`?
16. Suponha que `in` seja um objeto `Scanner` que é lido a partir de `System.in` e que seu programa chame

    ```
    String name = in.next();
    ```

    Qual é o valor de `name` se o usuário inserir `John Q. Public`?

Tópico avançado 4.6

Formatando números

O Tópico Avançado 4.6 mostra como controlar o número de dígitos depois do ponto de fração decimal ao imprimir um número utilizando o método `printf`. Isso é útil para mostrar um valor de moeda arredondado para dois dígitos, como em 0.30, em vez de 0.2975. Outras opções do método `printf` também são discutidas.

Tópico avançado 4.7

Utilizando caixas de diálogo para entrada e saída

O Tópico Avançado 4.7 mostra como utilizar caixas de diálogo para ler entrada ou exibir saída.

RESUMO DO CAPÍTULO

1. Java tem oito tipos primitivos, incluindo quatro tipos de inteiros e dois tipos de ponto flutuante.
2. Um cálculo numérico causa overflow se o resultado estiver fora do intervalo do tipo numérico.
3. Erros de arredondamento ocorrem quando uma conversão exata entre números não é possível.
4. Utilize uma coerção (*nomeDoTipo*) para converter um valor em um tipo diferente.
5. Utilize o método `Math.round` para arredondar um número de ponto flutuante para o inteiro mais próximo.
6. Uma variável `final` é uma constante. Depois de o valor ser configurado, ele não pode mais ser alterado.
7. Utilize constantes identificadas para facilitar a leitura e manutenção dos seus programas.
8. A atribuição para uma variável não é a mesma coisa que a igualdade matemática.
9. Os operadores `++` e `--` incrementam e decrementam uma variável.
10. Se os dois argumentos do operador `/` forem inteiros, o resultado será um inteiro e o resto será descartado.
11. O operador `%` calcula o resto de uma divisão.

12. A classe `Math` contém os métodos `sqrt` e `pow` para calcular raízes quadradas e potências.
13. Um método estático não opera sobre um objeto.
14. Uma string é uma seqüência de caracteres. Strings são objetos da classe `String`.
15. Strings podem ser concatenadas, isto é, ligadas uma a outra para produzir uma string nova e mais longa. A concatenação de string é indicada pelo operador `+`.
16. Sempre que um dos argumentos do operador `+` for uma string, o outro argumento será também convertido em uma string.
17. Se uma string contiver os dígitos de um número, utilize o método `Integer.parseInt` ou `Double.parseDouble` para obter o valor do número.
18. Utilize o método `substring` para extrair uma parte de uma string.
19. As posições das strings são contadas a partir de 0.
20. Utilize a classe `Scanner` para ler entrada de teclado em uma janela de console.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.io.PrintStream
   printf
java.lang.Double
   parseDouble
java.lang.Integer
   parseInt
   toString
   MAX_VALUE
   MIN_VALUE
java.lang.Math
   E
   PI
   abs
   acos
   asin
   atan
   atan2
   ceil
   cos
   exp
   floor
   log
   max
   min
   pow
   round
   sin
   sqrt
   tan
   toDegrees
   toRadians
java.lang.String
   format
   substring
java.lang.System
   in
java.math.BigDecimal
   add
   multiply
   subtract
java.math.BigInteger
   add
   multiply
   subtract
java.util.Scanner
   next
   nextDouble
   nextInt
   nextLine
javax.swing.JOptionPane
   showInputDialog
   showMessageDialog
```

EXERCÍCIOS DE REVISÃO

★★ **Exercício R4.1.** Escreva as expressões matemáticas a seguir em Java.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

$$G = 4\pi^2 \frac{a^3}{P^2 (m_1 + m_2)}$$

$$\text{FV} = \text{PV} \cdot \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}$$

$$c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}$$

★★ **Exercício R4.2.** Escreva em notação matemática as seguintes expressões em Java.

a. `dm = m * (Math.sqrt(1 + v / c) / (Math.sqrt(1 - v / c) - 1));`

b. `volume = Math.PI * r * r * h;`

c. `volume = 4 * Math.PI * Math.pow(r, 3) / 3;`

d. `p = Math.atan2(z, Math.sqrt(x * x + y * y));`

★★★ **Exercício R4.3.** O que há de errado com esta versão da fórmula quadrática?

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / 2 * a;
x2 = (-b + Math.sqrt(b * b - 4 * a * c)) / 2 * a;
```

★★ **Exercício R4.4.** Dê um exemplo de estouro de inteiro. O mesmo exemplo funcionaria corretamente se você utilizasse ponto flutuante?

★★ **Exercício R4.5.** Dê um exemplo de um erro de arredondamento de ponto flutuante. O mesmo exemplo funcionaria corretamente se você utilizasse inteiros e mudasse para uma unidade suficientemente pequena, como centavos, em vez de dólares, de modo que os valores não tivessem parte fracionária?

★★ **Exercício R4.6.** Considere o código a seguir:

```
CashRegister register = new CashRegister();
register.recordPurchase(19.93);
register.enterPayment(20, 0, 0, 0, 0);
System.out.print("Change: ");
System.out.println(register.giveChange());
```

O segmento de código imprime o total como `0.07000000000000028`. Explique por quê. Forneça uma recomendação para melhorar o código a fim de que os usuários não se confundam.

★ **Exercício R4.7.** Seja `n` um inteiro e `x` um número de ponto flutuante. Explique a diferença entre:

```
n = (int) x;
```

e

```
n = (int) Math.round(x);
```

★★★ **Exercício R4.8.** Seja `n` um inteiro e `x` um número de ponto flutuante. Explique a diferença entre:

```
n = (int) (x + 0.5);
```

e

```
n = (int) Math.round(x);
```

Para quais valores de `x` eles dão o mesmo resultado? Para quais valores de `x` eles dão resultados diferentes?

★ **Exercício R4.9.** Explique as diferenças entre `2`, `2.0`, `'2'`, `"2"` e `"2.0"`.

★ **Exercício R4.10.** Explique o que cada um dos dois segmentos de programa a seguir calcula:

```
int x = 2;
int y = x + x;
```

e

```
String s = "2";
String t = s + s;
```

★★ **Exercício R4.11.** Verdadeiro ou falso? (`x` é um `int` e `s` é uma `String`)

a. `Integer.parseInt("" + x)` é o mesmo que `x`

b. `"" + Integer.parseInt(s)` é o mesmo que `s`

c. `s.substring(0, s.length())` é o mesmo que `s`

★★ **Exercício R4.12.** Como você obtém o primeiro caractere de uma string? E o último? Como você remove o primeiro caractere? E o último?

★★★ **Exercício R4.13.** Como você obtém o último dígito de um inteiro? E o primeiro dígito? Isto é, se `n` for `23456`, como você descobre que o primeiro dígito é `2` e o último é `6`? Não converta o número em uma string. *Dica:* `%`, `Math.log`.

★★ **Exercício R4.14.** Este capítulo contém várias recomendações em relação a variáveis e constantes que tornam a leitura e a manutenção de programas mais fáceis. Resuma essas recomendações.

★★★ **Exercício R4.15.** O que é uma variável `final`? Você pode definir uma variável `final` sem fornecer seu valor? (Tente.)

★ **Exercício R4.16.** Quais são os valores das expressões a seguir? Em cada linha, suponha que:

```
double x = 2.5;
double y = -1.5;
int m = 18;
int n = 4;
String s = "Hello";
String t = "World";
```

a. `x + n * y - (x + n) * y`

b. `m / n + m % n`

c. `5 * x - n / 5`

d. `Math.sqrt(Math.sqrt(n))`

e. `(int) Math.round(x)`

f. `(int) Math.round(x) + (int) Math.round(y)`

g. `s + t`

h. `s + n`

i. `1 - (1 - (1 - (1 - (1 - n))))`

j. `s.substring(1, 3)`

k. `s.length() + t.length()`

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P4.1.** Aprimore a classe `CashRegister` adicionando métodos separados `enterDollars`, `enterQuarters`, `enterDimes`, `enterNickels` e `enterPennies`.

Utilize esta classe testadora:

```
public class CashRegisterTester
{
   public static void main (String[] args)
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(20.37);
      register.enterDollars(20);
      register.enterQuarters(2);
      System.out.println("Change: " + register.giveChange());
      System.out.println("Expected: 0.13");
   }
}
```

★ **Exercício P4.2.** Aprimore a classe `CashRegister` para que ela monitore o número total de itens em uma venda. Conte todas as compras registradas e forneça um método

```
int getItemCount()
```

que retorna o número de itens da compra atual. Lembre-se de redefinir a contagem no fim da compra.

★★ **Exercício P4.3.** Implemente uma classe `IceCreamCone` com os métodos `getSurfaceArea()` e `getVolume()`. No construtor, forneça a altura e o raio do cone. Tenha cuidado ao procurar a fórmula para a área da superfície – você só deve incluir a área externa em torno do cone, uma vez que o cone tem uma abertura no topo que segura o sorvete.

★★ **Exercício P4.4.** Escreva um programa que solicite ao usuário dois números e então imprime

- A soma
- A diferença
- O produto

- A média
- A distância (valor absoluto da diferença)
- O máximo (o maior dos dois)
- O mínimo (o menor dos dois)

Para fazer isso, implemente uma classe:

```
public class Pair
{
   /**
      Constrói um par.
      @param aFirst primeiro valor do par
      @param aSecond segundo valor do par
   */
   public Pair(double aFirst, double aSecond) { . . . }
   /**
      Calcula a soma dos valores desse par.
      @return soma dos primeiros e segundos valores
   */
   public double getSum() { . . . }
   . . .
}
```

Então implemente uma classe `PairTester` que cria um objeto `Pair`, invoca os métodos e imprime os resultados.

★ **Exercício P4.5.** Defina uma classe `DataSet` que calcula a soma e a média de uma seqüência de inteiros. Forneça os métodos

- `void addValue(int x)`
- `int getSum()`
- `double getAverage()`

Dica: Monitore a soma e a contagem dos valores.

Então, escreva um programa de teste `DataSetTester` que chama `addValue` quatro vezes e imprime os resultados reais e os esperados.

★★ **Exercício P4.6.** Escreva uma classe `DataSet` que calcula os maiores e menores valores em uma seqüência de números. Forneça os métodos

- `void addValue(int x)`
- `int getLargest()`
- `int getSmallest()`

Faça um registro do maior e do menor valor vistos até agora. Depois, utilize os métodos `Math.min` e `Math.max` para atualizá-los no método `addValue`. O que você deve utilizar como valores iniciais? *Dica:* `Integer.MIN_VALUE`, `Integer.MAX_VALUE`.

Escreva um programa de teste `DataSetTester` que chama `addValue` quatro vezes e imprime os resultados reais e os esperados.

★ **Exercício P4.7.** Escreva um programa que solicita ao usuário uma medida em metros e então a converte em milhas, pés e polegadas. Utilize uma classe

```
public class Converter
{
   /**
      Constrói um conversor que pode converter entre duas unidades.
      @param aConversionFactor fator pelo qual multiplicar
      para converter na unidade destino
   */
   public Converter(double aConversionFactor) { . . . }

   /**
      Converte a medida de origem na medida de destino.
      @param fromMeasurement medida
      @return valor de entrada convertido na unidade de destino
   */
   public double convertTo(double fromMeasurement) { . . . }

   /**
      Converte a medida de destino na medida de origem.
      @param toMeasurement medida de destino
      @return valor cuja conversão é a medida de destino
   */
   public double convertFrom(double toMeasurement) { . . . }
}
```

Na sua classe `ConverterTester`, construa e teste o objeto `Converter` a seguir:

```
final double MILE_TO_KM = 1.609;
Converter milesToMeters = new Converter(1000 * MILE_TO_KM);
```

★ **Exercício P4.8.** Escreva uma classe `Square` cujo construtor recebe o comprimento dos lados de um quadrado. Forneça então os métodos para calcular:

- A área e o perímetro do quadrado.
- O comprimento da diagonal (utilize o teorema de Pitágoras).

★★ **Exercício P4.9.** Implemente uma classe `SodaCan` cujo construtor recebe a altura e o diâmetro de uma lata de refrigerante. Forneça os métodos `getVolume` e `getSurfaceArea`. Forneça uma classe `SodaCanTester` que testa sua classe.

★★★ **Exercício P4.10.** Implemente uma classe `Balloon` que desenha um balão esférico que está sendo enchido com ar. O construtor cria um balão vazio. Forneça estes métodos:

- `void addAir(double amount)` adiciona a quantidade de ar dada
- `double getVolume()` obtém o volume atual
- `double getSurfaceArea()` obtém a área atual da superfície
- `double getRadius()` obtém o raio atual

Forneça uma classe `BalloonTester` que constrói um balão, adiciona 100 cm^3 de ar, testa os três métodos de acesso, adiciona outros 100 cm^3 de ar e testa os métodos de acesso novamente.

★★ **Exercício P4.11.** *Fornecendo troco.* Aprimore a classe `CashRegister` para que ela oriente um caixa bancário sobre como fornecer o troco. A caixa registradora calcula o valor a ser retornado ao cliente, em centavos.

Adicione os métodos a seguir à classe `CashRegister`:

- `int giveDollars()`
- `int giveQuarters()`
- `int giveDimes()`
- `int giveNickels()`
- `int givePennies()`

Cada método calcula o número de cédulas ou moedas de dólar a ser devolvido ao cliente e reduz o troco devido de acordo com o valor retornado. Você pode assumir que os métodos são chamados nesta ordem. Eis uma classe de teste:

```
public class CashRegisterTester
{
   public static void main(String[] args)
   {
      CashRegister register = new CashRegister();

      register.recordPurchase(8.37);
      register.enterPayment(10, 0, 0, 0, 0);
      System.out.println("Dollars: " + register.giveDollars());
      System.out.println("Expected: 1");
      System.out.println("Quarters: " + register.giveQuarters());
      System.out.println("Expected: 2");
      System.out.println("Dimes: " + register.giveDimes());
      System.out.println("Expected: 1");
      System.out.println("Nickels: " + register.giveNickels());
      System.out.println("Expected: 0");
      System.out.println("Pennies: " + register.givePennies());
      System.out.println("Expected: 3");
   }
}
```

★★★ **Exercício P4.12.** Escreva um programa que lê um número inteiro e o divide em uma seqüência de dígitos individuais na ordem inversa. Por exemplo, a entrada 16384 é exibida como:

```
4
8
3
6
1
```

Você pode assumir que a entrada não terá mais do que cinco dígitos e que não é negativa.

Defina uma classe `DigitExtractor`:

```
public class DigitExtractor
{
   /**
      Constrói um extrator de dígitos que obtém os dígitos
      de um inteiro na ordem inversa.
      @param anInteger inteiro a ser dividido em dígitos
```

```
      */
      public DigitExtractor(int anInteger) { . . . }

      /**
         Retorna o próximo dígito a ser extraído.
         @return próximo dígito
      */
      public int nextDigit() { . . . }
   }
```

Na sua classe principal, `DigitPrinter`, chame `System.out.println(myExtractor.nextDigit())` cinco vezes.

★★ **Exercício P4.13.** Implemente uma classe `QuadraticEquation` cujo construtor recebe os coeficientes `a`, `b`, `c` da equação quadrática $ax^2 + bx + c = 0$. Forneça os métodos `getSolution1` e `getSolution2` que mostram as soluções, utilizando a fórmula quadrática. Escreva uma classe de teste `QuadraticEquationTester` que cria um objeto `QuadraticEquation` e imprime as duas soluções.

★★★ **Exercício P4.14.** Escreva um programa que lê dois horários no formato militar (0900, 1730) e imprime o número de horas e minutos entre os dois horários. Eis uma execução de exemplo. A entrada de usuário está em cores.

```
Please enter the first time: 0900
Please enter the second time: 1730
8 hours 30 minutes
```

Você ganha um crédito extra se puder lidar com o caso em que o primeiro horário está adiantado em relação ao segundo horário:

```
Please enter the first time: 1730
Please enter the second time: 0900
15 hours 30 minutes
```

Implemente uma classe `TimeInterval` cujo construtor recebe dois horários militares. A classe deve ter dois métodos `getHours` e `getMinutes`.

★ **Exercício P4.15.** *Escrevendo letras grandes.* Uma letra `H` grande pode ser produzida desta maneira:

```
*   *
*   *
*****
*   *
*   *
```

Utilize a classe

```
public class LetterH
{
   public String toString()
   {
      return "*   *\n*   *\n*****\n*   *\n*   *\n";
   }
}
```

Defina classes semelhantes para as letras `E`, `L` e `O`. Então escreva a mensagem

```
H
E
L
L
O
```

em letras grandes.

★★ **Exercício P4.16.** Escreva uma classe `ChristmasTree` cujo método `toString` fornece uma string que representa uma árvore de Natal:

```
   /\
  /  \
 /    \
/      \
--------
  "  "
  "  "
  "  "
```

Lembre-se de utilizar seqüências de escape.

★★ **Exercício P4.17.** Seu trabalho é transformar os números `1`, `2`, `3`, `. . .`, `12` nos nomes dos meses `January`, `February`, `March`, `...`, `December` correspondentes. Implemente uma classe `Month` cujo parâmetro de construtor é o número do mês e cujo método `getName` retorna o nome do mês. *Dica:* Crie uma string bem longa `"January February March . . . "`, na qual você adiciona espaços de tal maneira que cada nome do mês tenha o mesmo comprimento. Utilize então `substring` para extrair o mês que você quer.

★★ **Exercício P4.18.** Escreva uma classe para calcular a data do domingo de Páscoa. O domingo de Páscoa é o primeiro domingo após a primeira lua cheia entre 22 de Março e 25 de Abril. Utilize este algoritmo, inventado em 1800, pelo matemático Carl Friedrich Gauss:

1. Seja `y` o ano (como 1800 ou 2001).
2. Divida `y` por `19` e chame o resto de `a`. Ignore o quociente.
3. Divida `y` por `100` para obter um quociente `b` e um resto `c`.
4. Divida `b` por `4` para obter um quociente `d` e um resto `e`.
5. Divida `8 * b + 13` por `25` para obter um quociente `g`. Ignore o resto.
6. Divida `19 * a + b - d - g + 15` por `30` para obter um resto `h`. Ignore o quociente.
7. Divida `c` por `4` para obter um quociente `j` e um resto `k`.
8. Divida `a + 11 * h` por `319` para obter um quociente `m`. Ignore o resto.
9. Divida `2 * e + 2 * j - k - h + m + 32` por `7` para obter um resto `r`. Ignore o quociente.
10. Divida `h - m + r + 90` por `25` para obter um quociente `n`. Ignore o resto.
11. Divida `h - m + r + n + 19` por `32` para obter um resto `p`. Ignore o quociente.

Então a Páscoa cai no dia `p` do mês `n`. Por exemplo, se `y` for `2001`:

```
a = 6
b = 20
```

```
c = 1
d = 5, e = 0
g = 6
h = 18
j = 0, k = 1
m = 0
r = 6
n = 4
p = 15
```

Portanto, em 2001, domingo de Páscoa caiu em 15 de abril. Escreva uma classe `Easter` com os métodos `getEasterSundayMonth` e `getEasterSundayDay`.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 4.1.** Neste projeto, você realizará cálculos com triângulos. Um triângulo é definido pelas coordenadas x e y dos seus três cantos.

Seu trabalho é calcular as seguintes propriedades de um dado triângulo:

- o comprimento de todos os lados
- o ângulo de todos os cantos
- o perímetro
- a área

Naturalmente, você deve implementar uma classe `Triangle` com os métodos apropriados. Forneça um programa que solicita a um usuário as coordenadas dos cantos e produza uma tabela elegantemente formatada das propriedades do triângulo.

Esse é um bom projeto de equipe para dois alunos. Ambos devem entrar em acordo sobre a interface `Triangle`. Um aluno implementa a classe `Triangle`, o outro implementa simultaneamente a interação com o usuário e a formatação.

★★★ **Projeto 4.2.** A classe `CashRegister` tem uma limitação lamentável: ela está estritamente associada ao sistema de moedas dos Estados Unidos e do Canadá. Pesquise o sistema utilizado na Europa. Seu objetivo é produzir uma caixa registradora que funciona com euros e centavos de euro. Em vez de projetar uma outra implementação `CashRegister` limitada para o mercado europeu, você deve projetar uma classe `Coin` separada e uma caixa registradora que possa funcionar com moedas de todos os tipos.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. `int` e `double`.
2. Quando a parte fracionária de `x` é ≥ 0.5.
3. Utilizando uma coerção: `(int) Math.round(x)`.
4. A primeira definição é utilizada dentro de um método, a segunda dentro de uma classe.

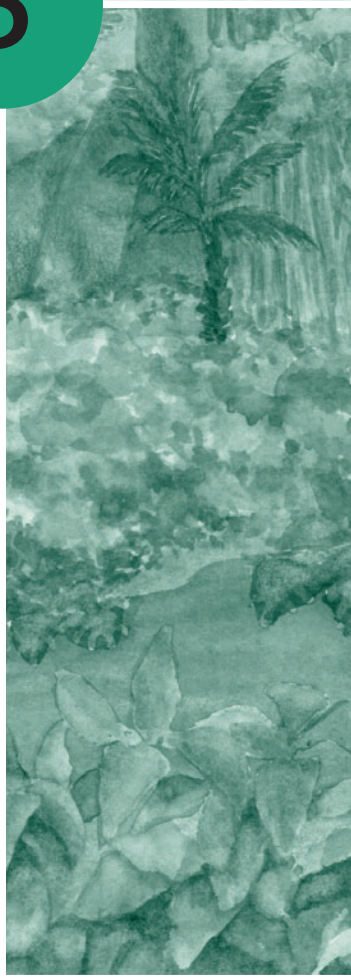
5. (1) Você deve utilizar uma constante identificada, não o "número mágico" 3.14. (2) 3.14 não é uma representação exata de π.
6. A instrução adiciona o valor `amount` à variável `balance`.
7. Um menos do que era antes.
8. 17 e 29.
9. Somente `s3` é dividido por 3. Para obter o resultado correto, use parênteses. Além disso, se `s1`, `s2` e `s3` são inteiros, você deve dividi-los por `3.0` para evitar a divisão de inteiro:

```
(s1 + s2 + s3) / 3.0
```

10. $\sqrt{x^2 + y^2}$
11. `x` é um número, não um objeto, e você não pode invocar métodos a partir de números.
12. Não – o método `println` é chamado no objeto `System.out`.
13. `s` é configurado como a string `Agent5`.
14. As strings `"i"` e `"ssissi"`.
15. A classe só tem um método para ler um único byte. Seria muito entediante formar caracteres, strings e números a partir desses bytes.
16. O valor é `"John"`. O método `next` lê a próxima *palavra.*

Capítulo 5

Decisões



Objetivos do capítulo

- Ser capaz de implementar decisões utilizando comandos `if`
- Entender como agrupar instruções em blocos
- Aprender a comparar inteiros, números de ponto flutuante, strings e objetos
- Reconhecer o ordenamento correto das decisões em múltiplos desvios
- Programar condições com variáveis e operadores booleanos

T Entender a importância da cobertura do teste

Os programas que vimos até agora conseguem fazer cálculos rápidos e exibir gráficos, mas são muito inflexíveis. Exceto quanto a variações na entrada, eles funcionavam da mesma maneira cada vez que eram executados. Um dos recursos essenciais dos programas de computador não-triviais é a capacidade de tomar decisões e executar diferentes ações, dependendo da natureza das entradas. O objetivo deste capítulo é aprender a programar decisões simples e complexas.

Conteúdo do Capítulo

5.1 O comando `if` 186
Sintaxe 5.1: O comando `if` 188
Sintaxe 5.2: Bloco de instruções 189
Dica de qualidade 5.1: Leiaute de chaves 189
Dica de produtividade 5.1: Recuos e tabulações 190
Tópico avançado 5.1: Operador de seleção ⊕

5.2 Comparando valores 191
Erro comum 5.1: Utilizando == para comparar strings 194
Dica de qualidade 5.2: Evite condições com efeitos colaterais 196

5.3 Múltiplas alternativas 196
Dica de produtividade 5.2: Atalhos de teclado para operações do mouse ⊕
Dica de produtividade 5.3: Copie e cole no editor ⊕
Tópico avançado 5.2: O comando `switch` ⊕
Erro comum 5.2: O problema do `else` oscilante 203
Dica de produtividade 5.4: Crie um cronograma e reserve tempo para problemas inesperados 204
Tópico avançado 5.3: Tipos enumerados ⊕
Sintaxe 5.3: Definindo um tipo enumerado ⊕

5.4 Utilizando expressões booleanas 205
Erro comum 5.3: Múltiplos operadores relacionais 207
Erro comum 5.4: Confundindo condições && com || 208
Tópico avançado 5.4: Avaliação preguiçosa dos operadores booleanos ⊕
Tópico avançado 5.5: Lei de De Morgan ⊕
Fato aleatório 5.1: Inteligência artificial ⊕

5.5T Cobertura do teste 210
Dica de qualidade 5.3: Calcule os dados de exemplo manualmente 211
Dica de qualidade 5.4: Prepare os casos de teste antecipadamente 212
Tópico avançado 5.6: Registro em log 212

5.1 O comando `if`

Programas de computador freqüentemente precisam tomar *decisões*, executando diferentes ações com base em uma condição.

Considere a classe de conta bancária do Capítulo 3. O método `withdraw` permite sacar quanto dinheiro você quiser. O saldo simplesmente fica negativo. Esse não é um modelo realista para uma conta bancária. Vamos implementar o método `withdraw` para que você não possa sacar mais dinheiro do que você tem na conta. Isto é, o método `withdraw` deve tomar uma *decisão*: permitir ou não saques.

O comando `if` permite a um programa executar diferentes ações com base em uma condição.

O comando `if` é utilizado para implementar uma decisão e tem duas partes: uma condição e um corpo. Se a *condição* for verdadeira, o *corpo* do comando será executado. O corpo do comando `if` consiste em uma instrução:

```
if (amount <= balance)
   balance = balance - amount;
```

A instrução de atribuição só é executada quando a quantia a ser sacada é menor que ou igual ao saldo (veja Figura 1).

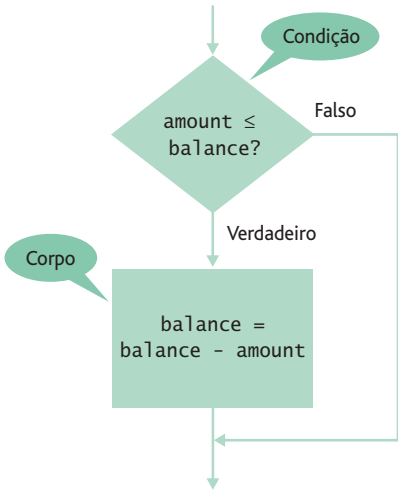


Figura 1
Fluxograma para um comando `if`.

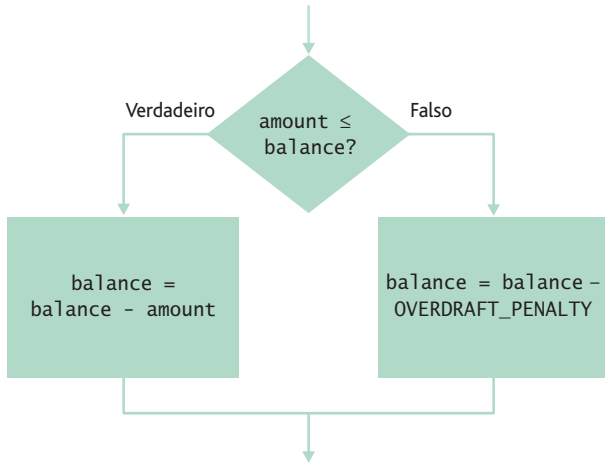


Figura 2
Fluxograma para um comando `if/else`.

Vamos tornar o método `withdraw` da classe `BankAccount` ainda mais realista. A maioria dos bancos não apenas proíbe saques que excedam o saldo na conta, como também cobra uma multa por cada tentativa de fazer isso.

Essa operação não pode ser simplesmente programada fornecendo dois comandos `if` complementares, como:

```
if (amount <= balance)
   balance = balance - amount;
if (amount > balance) // NÃO
   balance = balance - OVERDRAFT_PENALTY;
```

Há dois problemas nessa abordagem. Primeiro, se por alguma razão você precisar modificar a condição `amount <= balance`, você deverá lembrar de também atualizar a condição `amount > balance`. Se você não a atualizar, a lógica do programa não mais estará correta. Mais importante, se modificar o valor de `balance` no corpo do primeiro comando `if` (como nesse exemplo), então a segunda condição utilizará o novo valor.

Para implementar uma escolha entre alternativas, utilize a instrução `if/else`:

```
if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY;
```

Agora há somente uma condição. Se satisfeita, a primeira instrução é executada. Caso contrário, a segunda é executada. O fluxograma na Figura 2 fornece uma representação gráfica do comportamento do desvio.

Um bloco de instruções agrupa várias instruções.

Muito freqüentemente, porém, o corpo do comando `if` consiste em múltiplas instruções que devem ser executadas em seqüência sempre que a condição for verdadeira. Essas instruções devem estar agrupadas para formar um *bloco de instruções,* incluindo-as entre chaves `{ }`. Eis um exemplo.

```
if (amount <= balance)
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Uma instrução como

```
balance = balance - amount;
```

é chamada *instrução simples*. Uma instrução condicional como:

```
if (x >= 0) y = x;
```

é chamada *instrução composta*. No Capítulo 6, você encontrará comandos de laço (loop), que também são instruções compostas.

O corpo de um comando `if` ou a alternativa `else` deve ser uma instrução – isto é, uma instrução simples, uma instrução composta (por exemplo, outro comando `if`) ou um bloco de instruções.

SINTAXE 5.1 O comando `if`

```
if (condição)
   instrução
```

```
if (condição)
   instrução1
else
   instrução2
```

Exemplo:

```
if (amount <= balance)
   balance = balance - amount;
```

```
if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY;
```

Objetivo:

Executar uma instrução quando uma condição é verdadeira ou falsa

SINTAXE 5.2 Bloco de instruções

```
{
   instrução1
   instrução2
   . . .
}
```

Exemplo:

```
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Objetivo:

Agrupar várias instruções para formar uma única

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Por que utilizamos a condição `amount <= balance` e não `amount < balance` no exemplo da instrução `if/else`?
2. O que está logicamente errado com essa instrução

```
if (amount <= balance)
   newBalance = balance - amount; balance = newBalance;
```

e como corrigi-la?

DICA DE QUALIDADE 5.1

Leiaute de chaves

O posicionamento das chaves não é importante para o compilador, mas recomendamos veementemente que você siga uma regra simples: *Alinhe* { e }.

```
if (amount <= balance)
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Esse esquema facilita a identificação das chaves correspondentes.

Alguns programadores colocam a chave de abertura na mesma linha do `if`:

```
if (amount <= balance) {
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Isso economiza uma linha de código, mas torna mais difícil fazer a correspondência entre as chaves.

É importante que você selecione um esquema de leiaute e trabalhe com ele. O esquema que você escolher pode ser uma preferência pessoal ou um guia de estilo de codificação que você precisa seguir.

DICA DE PRODUTIVIDADE 5.1

Recuos e tabulações

Ao escrever programas Java, utilize recuos para indicar níveis de aninhamento:

```
public class BankAccount
{
|  . . .
|  public void withdraw(double amount)
|  {
|  |  if (amount <= balance)
|  |  {
|  |  |  double newBalance = balance - amount;
|  |  |  balance = newBalance;
|  |  }
|  }
|  . . .
}
0  1  2  3
```

Níveis de recuo (ou indentação)

Quantos espaços você deve utilizar por cada nível de recuo? Alguns programadores usam oito espaços por nível, mas essa não é uma boa escolha:

```
public class BankAccount
{
        . . .
        public void withdraw(double amount)
        {
                if (amount <= balance)
                {
                        double newBalance =
                                balance - amount;
                        balance = newBalance;
                }
        }
        . . .
}
```

Ela comprime demais o código no lado direito da tela. Como conseqüência, expressões longas freqüentemente precisam ser divididas em linhas separadas. Os valores mais comuns são dois, três ou quatro espaços por nível de recuo.

Como você move o cursor da coluna mais à esquerda para o nível de recuo apropriado? Uma estratégia perfeitamente razoável é pressionar a barra de espaço algumas vezes. Mas muitos programadores preferem utilizar a tecla de tabulação. O Tab move o cursor para a próxima parada de tabulação. Por padrão, há paradas de tabulação a cada oito colunas, mas a maioria dos editores permite alterar esse valor; você deve descobrir como configurar as paradas de tabulação do seu editor, digamos, a cada três colunas.

Alguns editores têm um recurso de *auto-recuo*. Eles inserem automaticamente o mesmo número de tabulações ou de espaços que o da linha anterior porque é bem provável que a nova linha pertença ao mesmo nível lógico de recuo. Se não, você deve adicionar ou remover uma tabulação, mas isso é mais rápido do que voltar todo o caminho até a margem esquerda.

Embora as tabulações para entrada de dados sejam boas, elas têm uma desvantagem: a impressão pode sair desordenada. Se imprimir um arquivo com tabulações, a impressora ou irá ignorar completamente todas as tabulações ou configurar paradas de tabulação a cada oito colunas. Portanto, é melhor salvar e imprimir seus arquivos com espaços do que com tabulações. A maioria dos editores tem configurações que convertem tabulações em espaços antes de você salvar ou imprimir um arquivo.

TÓPICO AVANÇADO 5.1

Operador de seleção

O Tópico Avançado 5.1 discute o operador de seleção `? :` para condições dentro de expressões.

5.2 Comparando valores

5.2.1 Operadores relacionais

Operadores relacionais comparam valores. O operador `==` testa a igualdade.

Um *operador relacional* testa o relacionamento entre dois valores. Um exemplo é o operador `<=` utilizado no teste

```
if (amount <= balance)
```

Java tem seis operadores relacionais:

Java	Notação matemática	Descrição
>	>	Maior que
>=	≥	Maior que ou igual a
<	<	Menor que
<=	≤	Menor que ou igual a
==	=	Igual
!=	≠	Diferente de

Como você pode ver, somente dois operadores relacionais (> e <) se parecem com aqueles da notação matemática que já conhecemos. Teclados de computador não têm teclas para ≥, ≤ ou ≠, mas os operadores `>=`, `<=` e `!=` são fáceis de lembrar porque são semelhantes.

Inicialmente, o operador `==` é confuso para a maioria dos iniciantes em Java. Em Java, o símbolo `=` já tem um significado, a saber, atribuição. O operador `==` denota teste de igualdade:

```
a = 5; // Atribui 5 a a
if (a == 5) . . . // Testa se a é igual a 5
```

Você precisará lembrar de utilizar == para teste de igualdade e = para atribuição.

5.2.2 Comparando números de ponto flutuante

Tenha cuidado ao comparar números de ponto flutuante para que possa tratar erros de arredondamento. Por exemplo, o código a seguir multiplica a raiz quadrada de 2 por ela mesma e então subtrai 2.

```
double r = Math.sqrt(2);
double d = r * r - 2;
if (d == 0)
   System.out.println("sqrt(2) squared minus 2 is 0");
else
   System.out.println(
         "sqrt(2) squared minus 2 is not 0 but " + d);
```

Embora as leis da matemática digam que $(\sqrt{2})^2 - 2$ é igual a 0, esse fragmento de programa imprime:

```
sqrt(2) squared minus 2 is not 0 but 4.440892098500626E-16
```

Infelizmente, esses erros de arredondamento são inevitáveis. Simplesmente não faz sentido, na maioria das vezes, comparar números de ponto flutuante de uma maneira exata. Em vez disso, teste se eles são *suficientemente próximos*.

> Ao comparar números de ponto flutuante, não teste a igualdade. Em vez disso, verifique se eles são suficientemente próximos.

Para testar se um número x é próximo de zero, você pode testar se o valor absoluto $|x|$ (isto é, o número com seu sinal removido) é menor que um número de limiar bem pequeno. O valor desse limiar costuma ser chamado de ε (a letra grega épsilon). É comum configurar ε como 10^{-14} ao testar números `double`.

Da mesma forma, você pode testar se dois números são aproximadamente iguais verificando se a diferença é próxima de 0.

$$|x - y| \leq \varepsilon$$

Em Java, programamos o teste assim:

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
   // x é aproximadamente igual a y
```

5.2.3 Comparando `strings`

Para testar se duas strings são iguais, você deve utilizar o método `equals`:

```
if (string1.equals(string2)) . . .
```

> Não utilize o operador == para comparar strings. Em vez disso, utilize o método `equals`.

Não utilize o operador == para comparar strings. A expressão

```
if (string1 == string2) // Não é útil
```

tem um significado diferente. Ele testa se as duas variáveis alfanuméricas referenciam o mesmo objeto string. Podemos ter strings

com conteúdos idênticos armazenados em objetos diferentes, portanto esse teste nunca faz sentido na programação real; veja o Erro comum 5.1.

Em Java, a distinção entre maiúsculas e minúsculas é importante. Por exemplo, `"Harry"` e `"HARRY"` não são strings idênticas. Para ignorar a distinção entre maiúsculas e minúsculas, utilize o método `equalsIgnoreCase`:

```
if (string1.equalsIgnoreCase(string2)) . . .
```

O método `compareTo` compara strings na ordem do dicionário.

Se duas strings não forem idênticas, ainda será recomendável entender o relacionamento entre elas. O método `compareTo` compara strings na ordem do dicionário. Se

```
string1.compareTo(string2) < 0
```

a string `string1` vem antes da string `string2` no dicionário. Por exemplo, este é o caso se `string1` for `"Harry"` e `string2` for `"Hello"`. Se

```
string1.compareTo(string2) > 0
```

`string1` vem depois de `string2` na ordem do dicionário. Por fim, se

```
string1.compareTo(string2) == 0
```

então `string1` e `string2` são iguais.

Na verdade, o ordenamento do tipo "dicionário" utilizado em Java é um pouco diferente daquele de um dicionário normal. Java diferencia letras maiúsculas de minúsculas e classifica caracteres colocando primeiro os números, em seguida os caracteres em letras maiúsculas e então os caracteres em letras minúsculas. Por exemplo, `1` vem antes de `B`, que vem antes de `a`. O caractere de espaço em branco vem antes de todos os outros caracteres.

Vamos investigar o processo de comparação bem de perto. Quando Java compara duas strings, as letras correspondentes são comparadas até uma das strings terminar ou até a primeira diferença ser encontrada. Se uma das strings terminar, a string mais longa será considerada a última. Se uma não-correspondência de caracteres for localizada, os caracteres são comparados para determinar qual string vem depois na seqüência do dicionário. Esse processo é chamado comparação lexicográfica. Por exemplo, vamos comparar `"car"` a `"cargo"`. As três primeiras letras correspondem e alcançamos o fim da primeira string. Portanto `"car"` vem antes de `"cargo"` na ordenação lexicográfica. Agora compare `"cathode"` com `"cargo"`. As duas primeiras letras correspondem. Na terceira posição de caractere, `t` vem depois de `r`, assim a string `"cathode"` vem depois de `"cargo"` na ordenação lexicográfica. (Veja Figura 3.)

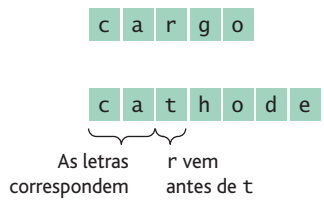


Figura 3
Comparação lexicográfica.

ERRO COMUM 5.1

Utilizando == para comparar strings

É um erro extremamente comum em Java escrever == quando o objetivo é usar `equals`. Isso é particularmente verdadeiro para strings. Se você escrever:

```
if (nickname == "Rob")
```

o teste então só será bem-sucedido se a variável `nickname` referenciar exatamente o mesmo objeto string que a constante string `"Rob"`. Por eficiência, Java cria apenas um objeto string para cada constante string. Portanto, o teste a seguir passará:

```
String nickname = "Rob";
. . .
if (nickname == "Rob") // O teste é verdadeiro
```

Mas se a string com as letras `R o b` for montada de alguma outra maneira, o teste falhará:

```
String name = "Robert";
String nickname = name.substring(0, 3);
. . .
if (nickname == "Rob") // O teste é falso
```

Essa é uma situação particularmente aflitiva: às vezes o código errado fará a coisa certa, outras vezes a coisa errada. Como objetos string sempre são construídos pelo compilador, nunca nos preocupamos se dois objetos string são compartilhados. Você deve lembrar de nunca utilizar == para comparar strings. Sempre utilize `equals` ou `compareTo` para comparar strings.

5.2.4 Comparando objetos

Se comparar duas referências de objeto com o operador ==, você testará se essas referências referenciam o mesmo objeto. Eis um exemplo:

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

A comparação

```
box1 == box2
```

é `true`. Ambas as variáveis de objeto referenciam o mesmo objeto. Mas a comparação

```
box1 == box3
```

é `false`. As duas variáveis de objeto referenciam objetos diferentes (ver a Figura 4). Não importa se os objetos tenham conteúdo idêntico.

O operador == testa se duas referências de objeto são idênticas. Para comparar o conteúdo de objetos, você precisa utilizar o método `equals`.

Você pode utilizar o método `equals` para testar se dois retângulos têm o mesmo conteúdo, isto é, se eles têm o mesmo canto superior esquerdo, a mesma largura e a mesma altura. Por exemplo, o teste

```
box1.equals(box3)
```

é verdadeiro.

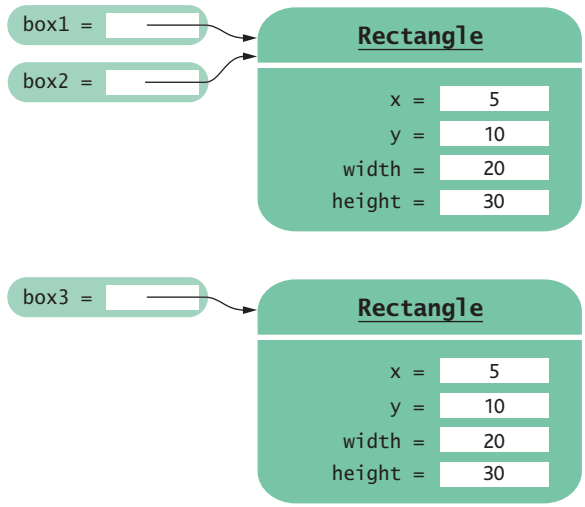


Figura 4 Comparando referências a objetos.

Entretanto, você deve ter cuidado ao utilizar o método `equals`. Ele só funciona corretamente se os implementadores da classe o definirem. A classe `Rectangle` tem um método `equals` adequado para comparar retângulos.

Nas suas próprias classes, você precisa fornecer um método `equals` apropriado. Você aprenderá a fazer isso no Capítulo 10. Até lá, você não deve utilizar o método `equals` para comparar objetos das suas próprias classes.

5.2.5 Testando null

A referência `null` não referencia nenhum objeto.

Uma referência a objeto pode ter o valor especial `null` se ela não referenciar nenhum objeto. É comum utilizar o valor `null` para indicar que um valor nunca foi configurado. Por exemplo,

```
String middleInitial = null; // Não configurado
if ( . . . )
   middleInitial = middleName.substring(0, 1);
```

Utilize o operador `==` (e não `equals`) para testar se uma referência a objeto é uma referência `null`:

```
if (middleInitial == null)
   System.out.println(firstName + " " + lastName);
else
   System.out.println(firstName + " " + middleInitial + ". " + lastName);
```

Observe que a referência `null` não é a mesma coisa que a string vazia `""`. A string vazia é uma string válida de comprimento 0, enquanto um `null` indica que uma variável do tipo string não referencia absolutamente nenhuma string.

Autoverificação da aprendizagem

3. Qual é o valor de `s.length()` se `s` for
   a. a string vazia `""`?
   b. a string `" "` contendo um espaço?
   c. `null`?
4. Quais comparações a seguir são sintaticamente incorretas? Quais delas são sintaticamente corretas, mas logicamente questionáveis?

```
String a = "1";
String b = "one";
double x = 1;
double y = 3 * (1.0 / 3);
```

   a. `a == "1"`
   b. `a == null`
   c. `a.equals("")`
   d. `a == b`
   e. `a == x`
   f. `x == y`
   g. `x - y == null`
   h. `x.equals(y)`

Dica de qualidade 5.2

Evite condições com efeitos colaterais

Em Java, é válido aninhar atribuições dentro de condições de teste:

```
if ((d = b * b - 4 * a * c) >= 0) r = Math.sqrt(d);
```

É válido utilizar o operador de decremento dentro de outras expressões:

```
if (n-- > 0) . . .
```

Esses dois usos são práticas de programação ruins, porque misturam um teste com uma outra atividade. A outra atividade (configurar a variável `d`, decrementando `n`) é chamada de *efeito colateral* do teste.

Como veremos no Tópico Avançado 6.2, condições com efeitos colaterais podem ser úteis ocasionalmente para simplificar laços; em comandos `if` elas devem ser evitadas sempre.

5.3 Múltiplas alternativas

5.3.1 Seqüências de comparações

Muitos cálculos requerem mais de uma única decisão `if/else`. Às vezes, você precisa criar uma série de comparações relacionadas.

Múltiplas condições podem ser combinadas para avaliar decisões complexas. O arranjo correto depende da lógica do problema a ser solucionado.

O programa a seguir solicita um valor que descreve a magnitude de um terremoto na escala Richter e imprime uma descrição do impacto provável do tremor. A escala Richter é a medida de intensidade de um terremoto. Cada medida na escala, por exemplo, de 6.0 a 7.0, significa um aumento de dez vezes na intensidade do tremor. O terremoto Loma Prieta de 1989, que danificou a Bay Bridge em San Francisco e destruiu muitos edifícios em várias cidades nessa área, registrou 7.1 na escala Richter.

ch05/quake/Earthquake.java

```
1  /**
2     Uma classe que descreve os efeitos de um terremoto.
3  */
4  public class Earthquake
5  {
6     /**
7        Constrói um objeto Earthquake.
8        @param magnitude magnitude na escala Richter
9     */
10    public Earthquake(double magnitude)
11    {
12       richter = magnitude;
13    }
14
15    /**
16       Obtém uma descrição do efeito do terremoto.
17       @return descrição do efeito
18    */
19    public String getDescription()
20    {
21       String r;
22       if (richter >= 8.0)
23          r = "Most structures fall";
24       else if (richter >= 7.0)
25          r = "Many buildings destroyed";
26       else if (richter >= 6.0)
27          r = "Many buildings considerably damaged, some collapse";
28       else if (richter >= 4.5)
29          r = "Damage to poorly constructed buildings";
30       else if (richter >= 3.5)
31          r = "Felt by many people, no destruction";
32       else if (richter >= 0)
33          r = "Generally not felt by people";
34       else
35          r = "Negative numbers are not valid";
36       return r;
37    }
38
39    private double richter;
40 }
```

ch05/quake/EarthquakeRunner.java

```
1  import java.util.Scanner;
2
3  /**
4     Esse programa imprime uma descrição de um terremoto de uma dada magnitude.
5  */
6  public class EarthquakeRunner
7  {
8     public static void main(String[] args)
9     {
10       Scanner in = new Scanner(System.in);
11
12       System.out.print("Enter a magnitude on the Richter scale: ");
13       double magnitude = in.nextDouble();
14       Earthquake quake = new Earthquake(magnitude);
15       System.out.println(quake.getDescription());
16    }
17 }
```

Saída

```
Enter a magnitude on the Richter scale: 7.1
Many buildings destroyed
```

Aqui devemos classificar as condições e testar contra o primeiro limite superior. Suponha que a ordem dos testes seja invertida:

```
if (richter >= 0) // Testes na ordem errada
   r = "Generally not felt by people";
else if (richter >= 3.5)
   r = "Felt by many people, no destruction";
else if (richter >= 4.5)
   r = "Damage to poorly constructed buildings";
else if (richter >= 6.0)
   r = "Many buildings considerably damaged, some collapse";
else if (richter >= 7.0)
   r = "Many buildings destroyed";
else if (richter >= 8.0)
   r = "Most structures fall";
```

Isso não funciona. Todos os valores não-negativos de `richter` entram no primeiro caso e os outros testes nunca serão tentados.

Nesse exemplo, também é importante utilizar um teste `if/else/else`, não apenas múltiplos comandos `if` independentes. Considere esta seqüência de testes independentes:

```
if (richter >= 8.0) // Não utilizou else
   r = "Most structures fall";
if (richter >= 7.0)
   r = "Many buildings destroyed";
if (richter >= 6.0)
   r = "Many buildings considerably damaged, some collapse";
```

```
if (richter >= 4.5)
   r = "Damage to poorly constructed buildings";
if (richter >= 3.5)
   r = "Felt by many people, no destruction";
if (richter >= 0)
   r = "Generally not felt by people";
```

Agora as alternativas não são mais exclusivas. Se `richter` for 6.0, então todos os últimos quatro testes corresponderão e `r` será configurado quatro vezes.

DICA DE PRODUTIVIDADE 5.2

Atalhos de teclado para operações do mouse

A Dica de Produtividade 5.2 mostra como atalhos pelo teclado podem acelerar o uso de comandos comuns.

DICA DE PRODUTIVIDADE 5.3

Copie e cole no editor

A Dica de Produtividade 5.3 mostra como utilizar copiar e colar para minimizar digitação repetitiva.

TÓPICO AVANÇADO 5.2

O comando `switch`

O Tópico Avançado 5.2 discute o comando `switch`, uma alternativa à seqüência das instruções `if/else/else` em que todas as condições testam o mesmo valor contra constantes.

5.3.2 Desvios aninhados

Alguns cálculos têm múltiplos *níveis* de tomada de decisão. Primeiro você toma uma decisão e cada um dos resultados leva a uma outra decisão. Eis um exemplo típico.

Nos Estados Unidos, contribuintes pagam imposto de renda federal de acordo com seus respectivos rendimentos e estado civil. Há dois tipos principais de declaração de imposto: uma para contribuintes solteiros e outra para contribuintes casados que "declaram em conjunto", o que significa que eles somam seus rendimentos e pagam impostos sobre o total. (Na realidade, há dois outros tipos de declaração, "chefe de família" e "declaração separado de casados", que ignoraremos aqui para simplificar.) A Tabela 1 fornece os cálculos da faixa de imposto para cada um dos tipos de declaração, utilizando os valores de 1992 para a restituição de imposto federal. (Estamos utilizando o sistema de declaração de imposto de renda norte-americano de 1992 nesta ilustração devido à sua simplicidade. A legislação de 1993 aumentou as alíquotas em cada uma

Tabela 1 Sistema de declaração de imposto de renda norte-americano (1992)

Se sua declaração for do tipo Solteiro		Se sua declaração for do tipo Casado	
Faixa do imposto	Percentual	Faixa do imposto	Percentual
US$0 . . . $21,450	15%	US$0 . . . US$35,800	15%
Rendimentos acima de US$ 21,450, até US$ 51,900	28%	Rendimentos acima de US$ 35,800, até US$ 86,500	28%
Rendimentos acima de US$ 51,900	31%	Rendimentos acima de US$ 86,500	31%

das categorias e acrescentou regras mais complexas. No momento em que você estiver lendo isto, a legislação tributária pode muito bem ter mudado novamente e ser ainda mais complexa.)

Agora vamos calcular os impostos devidos, dado o tipo de declaração e um rendimento. Primeiro, devemos dividir o tipo de declaração. Então, para cada tipo, precisamos ter outro desvio de acordo com o nível de renda.

O processo de dois níveis de decisão é refletido em dois níveis de comandos `if`. Dizemos que o teste da renda é *aninhado* dentro do teste do tipo de declaração. (Veja um fluxograma na Figura 5.)

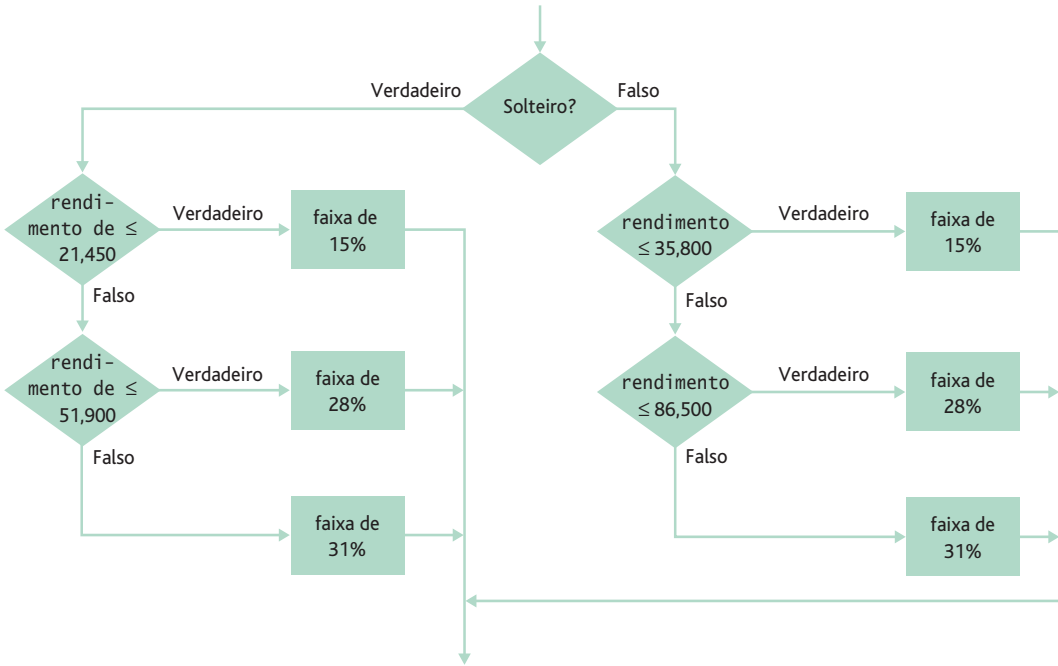


Figura 5 Cálculo do imposto de renda utilizando as regras de 1992.

ch05/tax/TaxReturn.java

```
/**
   Uma restituição de imposto de um contribuinte em 1992.
*/
public class TaxReturn
{
   /**
      Constrói um objeto TaxReturn para uma dada renda e
      estado civil.
      @param anIncome renda do contribuinte
      @param aStatus ou SINGLE ou MARRIED
   */
   public TaxReturn(double anIncome, int aStatus)
   {
      income = anIncome;
      status = aStatus;
   }

   public double getTax()
   {
      double tax = 0;

      if (status == SINGLE)
      {
         if (income <= SINGLE_BRACKET1)
            tax = RATE1 * income;
         else if (income <= SINGLE_BRACKET2)
            tax = RATE1 * SINGLE_BRACKET1
                  + RATE2 * (income - SINGLE_BRACKET1);
         else
            tax = RATE1 * SINGLE_BRACKET1
                  + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1);
                  + RATE3 * (income - SINGLE_BRACKET2);
      }
      else
      {
         if (income <= MARRIED_BRACKET1)
            tax = RATE1 * income;
         else if (income <= MARRIED_BRACKET2)
            tax = RATE1 * MARRIED_BRACKET1
                  + RATE2 * (income - MARRIED_BRACKET1);
         else
            tax = RATE1 * MARRIED_BRACKET1
                  + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1);
                  + RATE3 * (income - MARRIED_BRACKET2);
      }

      return tax;
   }

   public static final int SINGLE = 1;
   public static final int MARRIED = 2;

   private static final double RATE1 = 0.15;
   private static final double RATE2 = 0.28;
```

```
55      private static final double RATE3 = 0.31;
56
57      private static final double SINGLE_BRACKET1 = 21450;
58      private static final double SINGLE_BRACKET2 = 51900;
59
60      private static final double MARRIED_BRACKET1 = 35800;
61      private static final double MARRIED_BRACKET2 = 86500;
62
63      private double income;
64      private int status;
65   }
```

ch05/tax/TaxCalculator.java

```
1   import java.util.Scanner;
2
3   /**
4      Este programa calcula uma restituição de imposto simples.
5   */
6   public class TaxCalculator
7   {
8      public static void main(String[] args)
9      {
10        Scanner in = new Scanner(System.in);
11
12        System.out.print("Please enter your income: ");
13        double income = in.nextDouble();
14
15        System.out.print("Are you married? (Y/N) ");
16        String input = in.next();
17        int status;
18        if (input.equalsIgnoreCase("Y"))
19           status = TaxReturn.MARRIED;
20        else
21           status = TaxReturn.SINGLE;
22        TaxReturn aTaxReturn = new TaxReturn(income, status);
23
24        System.out.println("Tax: "
25              + aTaxReturn.getTax());
26     }
27  }
```

Saída

```
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
```

Autoverificação da aprendizagem

5. O comando `if/else/else` para a intensidade do terremoto primeiro testou os valores mais altos e depois os valores mais baixos. Podemos inverter essa ordem?

6. Algumas pessoas são contra faixas mais altas de imposto que incidem sobre rendimentos mais altos, argumentando que, no final, você poderia ter *menos* dinheiro depois de pagar os impostos quando você consegue um aumento salarial. Qual é a falha nesse argumento?

ERRO COMUM 5.2

O problema do `else` oscilante

Quando uma instrução `if` é aninhada dentro de uma outra instrução `if`, o seguinte erro pode ocorrer.

```
if (richter >= 0)
   if (richter <= 4)
      System.out.println("The earthquake is harmless");
else   // Armadilha!
   System.out.println("Negative value not allowed");
```

O nível de recuo parece sugerir que o `else` está agrupado com o teste `richter >= 0`. Infelizmente, esse não é o caso. O compilador ignora todos os recuos e segue a regra de que um `else` sempre pertence ao `if` mais próximo, como aqui:

```
if (richter >= 0)
   if (richter <= 4)
      System.out.println("The earthquake is harmless");
   else   // Armadilha!
      System.out.println("Negative value not allowed");
```

Não é isso o que queríamos. Queríamos agrupar o `else` com o primeiro `if`. Para fazer isso, precisamos utilizar chaves.

```
if (richter >= 0)
{
   if (richter <= 4)
      System.out.println("The earthquake is harmless");
}
else
   System.out.println("Negative value not allowed");
```

Para evitar ter de pensar no pareamento do `else`, recomendamos que você *sempre* utilize um conjunto de chaves quando o corpo de um `if` contém outro `if`. No exemplo a seguir, as chaves não são estritamente necessárias, mas ajudam a tornar o código mais legível:

```
if (richter >= 0)
{
   if (richter <= 4)
      System.out.println("The earthquake is harmless");
   else
      System.out.println("Damage may occur");
}
```

O `else` ambíguo é chamado de `else` oscilante e tanto é um defeito sintático que alguns projetistas de linguagens de programação desenvolveram uma sintaxe melhorada que o evita completamente. Por exemplo, o Algol 68 usa a construção:

`if` *condição* `then` *instrução* `else` *instrução* `fi;`

A parte `else` é opcional, mas, como o final da instrução `if` está claramente marcado, o agrupamento não é ambíguo se houver dois `if`s e somente um `else`. Eis os dois casos possíveis:

```
if c1 then if c2 then s1 else s2 fi fi;
if c1 then if c2 then s1 fi else s2 fi;
```

A propósito, `fi` é apenas `if` de trás para frente. Outras linguagens usam `endif`, que tem o mesmo propósito, mas é menos divertido.

DICA DE PRODUTIVIDADE 5.4

Crie um cronograma e reserve tempo para problemas inesperados

É notório o fato de que softwares comerciais nunca são distribuídos na data prometida. Por exemplo, a Microsoft originalmente prometeu que o sucessor do sistema operacional Windows XP estaria disponível em 2004, em seguida no começo de 2005 e então no final de 2005. Algumas promessas iniciais talvez não tenham sido realistas. É do interesse da Microsoft deixar que potenciais clientes tenham uma previsão da disponibilidade iminente do produto para que, nesse meio tempo, não troquem por um produto diferente. Inegavelmente, porém, a Microsoft não antecipou toda a complexidade das tarefas que ela mesma deveria resolver.

A Microsoft pode postergar a entrega do seu produto, mas, talvez, você não possa fazer isso. Como um aluno ou um programador, espera-se que você gerencie seu tempo de forma inteligente e termine suas tarefas dentro do prazo. Provavelmente você consegue fazer testes simples de programação na noite antes do prazo de entrega, mas uma tarefa que talvez seja duas vezes mais difícil pode muito bem demorar quatro vezes mais, porque mais coisas podem dar errado. Portanto, você deve criar um cronograma sempre que iniciar um projeto de programação.

Primeiro, estime de forma realista quanto tempo você levará para:

- Projetar a lógica do programa
- Desenvolver casos de teste
- Digitar o programa e corrigir erros de sintaxe
- Testar e depurar o programa

Por exemplo, para o programa de imposto de renda eu poderia estimar 30 minutos para o projeto, porque a maior parte já foi feita; 30 minutos para desenvolver casos de teste; uma hora para a entrada de dados e correção de erros de sintaxe e 2 horas para testar e depurar. Isso dá um total de 4 horas. Se eu trabalhar 2 horas por dia nesse projeto, ele levará dois dias.

Pense então nas coisas que podem dar errado. Seu computador pode apresentar algum problema. Talvez não haja máquinas livres no laboratório. Talvez um problema com o sistema computacional seja difícil de solucionar. (Essa é uma preocupação especialmente importante para iniciantes. É *muito* comum perder um dia com um problema trivial porque simplesmente demora para encontrar uma pessoa que conheça o comando "mágico" para que esse problema possa ser solucionado.) Como regra geral, *dobre* o tempo estimado. Isto é, você deve iniciar quatro dias, não dois dias, antes da data de entrega. Se nada der errado, excelente; você terminou o programa dois dias antes. Quando ocorre aquele problema inevitável, você tem um tempo extra que o protege contra constrangimentos e falhas.

TÓPICO AVANÇADO 5.3

Tipos enumerados

O Tópico Avançado 5.3 introduz tipos enumerados – tipos que contêm um número finito de valores. Um exemplo de um tipo enumerado é um tipo `FilingStatus` com valores `SINGLE` e `MARRIED`. Isso é mais seguro do que usar valores inteiros, como fizemos na classe `TaxReturn`.

5.4 Utilizando expressões booleanas

5.4.1 O tipo `boolean`

Em Java, uma expressão como `amount < 1000` tem um valor, assim como a expressão `amount + 1000` também tem um valor. O valor de uma expressão relacional é `true` ou `false`. Por exemplo, se `amount` for 500, então o valor de `amount < 1000` será `true`. Teste: O fragmento de programa

```
double amount = 0;
System.out.println(amount < 1000);
```

O tipo `boolean` tem dois valores: `true` e `false`.

imprime `true`. Os valores `true` e `false` não são números, nem objetos de uma classe. Eles pertencem a um tipo separado, chamado `boolean`. O tipo booleano recebe este nome em homenagem ao matemático George Boole (1815–1864), um pioneiro no estudo da lógica.

5.4.2 Métodos de predicado

Um *método de predicado* é um método que retorna um valor `boolean`. Eis um exemplo:

Um método de predicado retorna um valor `boolean`.

```
public class BankAccount
{
   public boolean isOverdrawn()
   {
      return balance < 0;
   }
}
```

Você pode utilizar o valor de retorno desse método como a condição de uma instrução `if`:

```
if (harrysChecking.isOverdrawn()) . . .
```

Há vários métodos de predicado estáticos úteis na classe `Character`:

```
isDigit
isLetter
isUpperCase
isLowerCase
```

que permitem testar se um caractere é um dígito, uma letra, uma letra maiúscula ou uma letra minúscula:

```
if (Character.isUpperCase(ch)) . . .
```

É uma convenção comum atribuir o prefixo "`is`" ou "`has`" ao nome de um método de predicado.

A classe `Scanner` contém métodos de predicado úteis para testar se a próxima entrada será bem-sucedida. O método `hasNextInt` retornará `true` se a próxima seqüência de caracteres denotar um inteiro. Uma boa idéia é chamar esse método antes de chamar `nextInt`:

```
if (in.hasNextInt()) n = in.nextInt();
```

De maneira semelhante, o método `hasNextDouble` testa se uma chamada a `nextDouble` será bem-sucedida.

5.4.3 Os operadores booleanos

Suponha que você queira descobrir se `amount` está entre 0 e 1000. Então duas condições têm de ser verdadeiras: `amount` deve ser maior que 0 *e* deve ser menor que 1000. Em Java utilizamos o operador `&&` para representar *e* combinar condições de teste. Isto é, você pode escrever o teste desta maneira:

```
if (0 < amount && amount < 1000) . . .
```

Você pode desenvolver testes complexos com os operadores booleanos `&&` (e), `||` (ou) e `!` (não).

O operador `&&` combina vários testes em um novo teste que só será verdadeiro quando todas as condições são verdadeiras. Um operador que combina condições de teste é chamado *operador lógico*.

O operador lógico `||` (*ou*) também combina duas ou mais condições. O teste resultante será bem-sucedido se pelo menos uma das condições for verdadeira. Por exemplo, eis um teste para verificar se a string `input` é um `"S"` ou um `"M"`:

```
if (input.equals("S") || input.equals("M")) . . .
```

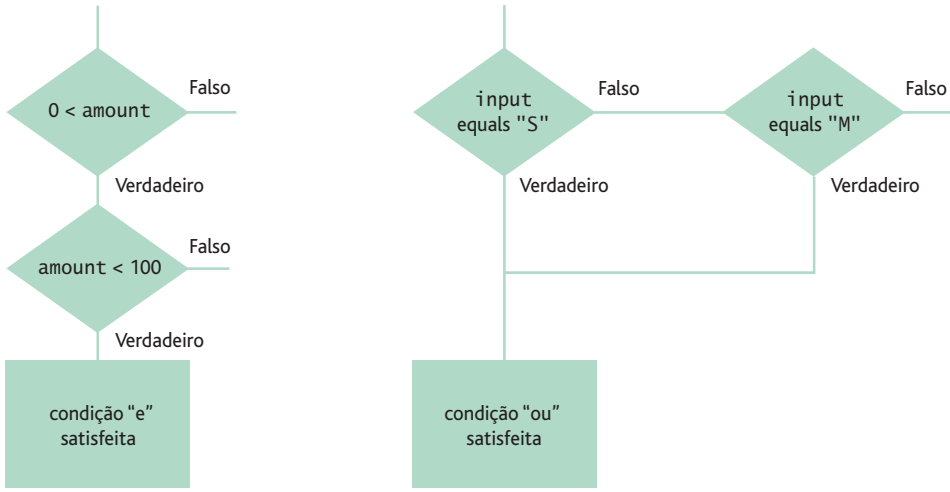


Figura 6 Fluxogramas para combinações `&&` e `||`.

A Figura 6 mostra os fluxogramas para esses exemplos.

Às vezes você precisa *reverter* uma condição com o operador lógico `!` (*não*). Por exemplo, podemos querer executar certa ação somente se duas strings *não* forem iguais:

```
if (!input.equals("S")) . . .
```

O operador `!` recebe uma única condição e é avaliado para `true` se essa condição for falsa e para `false` se a condição for verdadeira.

Eis um resumo das três operações lógicas:

A	B	A && B
true	true	true
true	false	false
false	*Qualquer*	false

A	B	A || B
true	*Qualquer*	true
false	true	true
false	false	false

A	!A
true	false
false	true

ERRO COMUM 5.3

Múltiplos operadores relacionais

Considere a expressão:

```
if (0 < amount < 1000) . . . // Erro
```

Isso faz com que a notação matemática para "`amount` pareça estar entre 0 e 1000". Mas em Java, é um erro de sintaxe.

Vamos dissecar a condição. A primeira metade, `0 < amount`, é um teste com resultado `true` ou `false`. O resultado desse teste (`true` ou `false`) é então comparado com 1000. Isso parece não fazer sentido. `true` é maior que 1000 ou não? Podemos comparar valores e números de verdade? Em Java, não. O compilador Java rejeita essa instrução.

Em vez disso, utilize && para combinar dois testes separados:

```
if (0 < amount && amount < 1000) . . .
```

Um outro erro comum, ao longo das mesmas linhas, é escrever:

```
if (ch == 'S' || 'M') . . . // Erro
```

para testar se `ch` é `'S'` ou `'M'`. Novamente, o compilador Java marca essa construção como um erro. Você não pode aplicar o operador `||` a caracteres. Você precisa escrever duas expressões booleanas e associá-las com o operador `||`:

```
if (ch == 'S' || ch == 'M') . . .
```

ERRO COMUM 5.4

Confundindo condições && com ||

É um erro incrivelmente comum confundir as condições *e* e *ou*. Um valor reside entre 0 e 100 se ele for pelo menos 0 *e* no máximo 100. Ele reside fora desse intervalo se for menor que 0 *ou* maior que 100. Não há nenhuma regra de ouro; você só precisa pensar cuidadosamente.

Freqüentemente o *e* ou o *ou* é claramente declarado e, portanto, não é muito difícil implementá-los. Às vezes, porém, o texto não é tão explícito. É bem comum que condições individuais estejam perfeitamente separadas em uma lista com marcadores, mas com pouca indicação de como devem ser combinadas. As instruções para a restituição do imposto de 1992 informam que você pode usar uma declaração de solteiro se um dos seguintes for verdadeiro:

- Você nunca foi casado.
- Você estava juridicamente separado ou se divorciou em 31 de dezembro de 1992.
- Você ficou viúvo antes de 1º de janeiro de 1992 e não se casou novamente em 1992.

Como o teste passa se *uma* das condições for verdadeira, você precisa combinar as condições com *ou*. Além disso, as mesmas instruções afirmam que você pode utilizar a declaração mais vantajosa de casado se todas as condições a seguir forem verdadeiras:

- Seu cônjuge morreu em 1990 ou 1991 e você não se casou novamente em 1992.
- Você tem um filho declarado como dependente.
- Esse filho viveu com você todo o ano de 1992.
- Você arca apenas com a metade do custo das despesas com esse filho.
- Você preencheu (ou poderia ter preenchido) uma restituição em conjunto com seu cônjuge no ano em que ele ou ela morreu.

Como *todas* as condições devem ser verdadeiras para que o teste passe, você precisa combiná-las com um *e*.

TÓPICO AVANÇADO 5.4

Avaliação preguiçosa dos operadores booleanos

O Tópico Avançado 5.4 explica a avaliação "preguiçosa" dos operadores booleanos: o fato de o lado direito das expressões && e || não ser avaliado se o lado esquerdo já determinar o resultado.

TÓPICO AVANÇADO 5.5

Lei de De Morgan

O Tópico Avançado 5.5 abrange a lei de De Morgan, uma lei da lógica utilizada para simplificar condições em que operadores *não* são aplicados a expressões *e/ou*.

5.4.4 Usando variáveis booleanas

Você pode utilizar uma variável booleana se souber que há apenas dois valores possíveis. Examine mais uma vez o programa de imposto na Seção 5.3.2. O estado civil é solteiro ou casado. Em vez de utilizar um inteiro, você pode utilizar uma variável do tipo `boolean`:

```
private boolean married;
```

> Você pode armazenar o resultado de uma condição em uma variável booleana.

A vantagem é que você não armazena acidentalmente um terceiro valor na variável.

Essa variável booleana pode então ser utilizada em um teste:

```
if (married)
   . . .
else
   . . .
```

Às vezes, variáveis booleanas são chamadas *flags* (bandeiras) porque só podem ter dois estados: "hasteada" (ativada) ou "não-hasteada" (desativada).

Vale a pena pensar cuidadosamente sobre a atribuição de nomes às variáveis booleanas. No nosso exemplo, não seria uma boa idéia atribuir o nome `maritalStatus` à variável booleana. O que significa o estado civil ser `true`? Com um nome como `married` não há ambigüidade; se `married` é `true`, o contribuinte é casado.

A propósito, é considerado inapropriado escrever um teste como

```
if (married == true) . . . // Não faça isso
```

Simplesmente utilize o teste mais simples

```
if (married) . . .
```

No Capítulo 6 utilizaremos variáveis booleanas para controlar laços complexos.

Autoverificação da aprendizagem

7. Quando a instrução

```
System.out.println(x > 0 || x < 0);
```

imprime `false`?

8. Reescreva a expressão a seguir, evitando a comparação com `false`:

```
if (Character.isDigit(ch) == false) . . .
```

Fato aleatório 5.1

Inteligência artificial

O Fato Aleatório 5.1 discute a inteligência artificial, processos de computador que parecem simular o raciocínio humano inteligente.

5.5 Cobertura do teste

O teste da caixa preta descreve um método de teste que não leva em consideração a estrutura da implementação.

O teste da caixa preta usa as informações sobre a estrutura de um programa.

A cobertura do teste é uma medida de quantas partes de um programa foram testadas.

Testar a funcionalidade de um programa sem considerar a estrutura interna é chamado de *teste da caixa preta*. Essa é uma parte importante do teste, pois, afinal de contas, os usuários de um programa não conhecem sua estrutura interna. Se um programa funciona perfeitamente com todas as entradas, então seguramente faz seu trabalho.

Mas é impossível assegurar que um programa funcionará corretamente para todas as entradas se você fornecer apenas um número finito de casos de teste. Como o famoso cientista da computação Edsger Dijkstra apontou, o teste só pode mostrar a presença de problemas – não a ausência deles. Para ganhar mais confiança na correção de um programa, é útil pensar na estrutura interna. Estratégias de teste que investigam a parte interna de um programa são chamadas *teste da caixa preta*. Realizar testes de unidade em cada método é uma parte do teste da caixa preta.

Você quer certificar-se de que cada parte do seu programa seja testada pelo menos uma vez por um dos seus casos de teste. Isso é chamado de *teste de cobertura*. Se algum código nunca for executado por um dos seus casos de teste, você não terá como saber se ele teria um desempenho correto se fosse executado pela entrada do usuário. Isso significa que você precisa examinar todos os desvios do `if/else` para ver se cada um deles é coberto por algum caso de teste. Há muitos desvios condicionais no código apenas para gerenciar entradas estranhas e anormais, mas eles ainda fazem algo mais. É um fenômeno comum que eles acabem fazendo algo incorretamente, mas essas falhas nunca são descobertas durante o teste, porque ninguém forneceu entradas estranhas ou anormais. Naturalmente, essas falhas tornam-se aparentes de forma imediata quando o programa é distribuído e o primeiro usuário digita uma entrada incomum e fica furioso quando o programa comporta-se de uma maneira inconveniente. O correto é assegurar que cada fragmento do código seja coberto por algum caso de teste.

Por exemplo, ao testar o método `getTax` da classe `TaxReturn`, você quer certificar-se de que cada instrução `if` seja coberta por pelo menos um caso de teste. Você deve testar tanto os contribuintes solteiros como os casados, com rendas em cada uma das faixas de imposto.

Ao selecionar casos de teste, você deve ter o hábito de incluir *casos de teste de limite*: valores válidos que residem no limite do conjunto de entradas aceitáveis.

Casos de teste de limite são os casos de teste que estão no limite das entradas aceitáveis.

Por exemplo, o que acontece quando você calcula os impostos sobre uma renda de 0 ou se uma conta bancária tem uma taxa de juros de 0%? Casos limite continuam sendo entradas legítimas e espera-se que o programa trate-os corretamente – freqüentemente de forma trivial ou por meio de casos especiais. Testar casos limite é importante, porque programadores costumam cometer erros ao lidar com condições limite. Dividir por zero, extrair caracteres de strings vazias e acessar ponteiros nulos são sintomas comuns de erros de limite.

Autoverificação da aprendizagem

9. Quantos casos de teste você precisa para abranger todos os desvios do método `getDescription` da classe `Earthquake`?
10. Forneça um caso de teste de limites para o programa `EarthquakeRunner`. Qual saída você espera?

Dica de qualidade 5.3

Calcule dados de exemplo manualmente

Você deve calcular casos de teste manualmente para garantir que seu aplicativo calcula a resposta correta.

Normalmente é difícil ou impossível provar que um dado programa funciona corretamente em todos os casos. Para ganhar confiança quanto à exatidão de um programa ou para entender por que ele não funciona como deveria, dados de exemplo calculados manualmente são imprescindíveis. Se o programa chegar aos mesmos resultados que os do cálculo manual, nossa confiança será maior. Se os resultados manuais diferirem dos resultados do programa, temos um ponto de partida para o processo de depuração.

Surpreendentemente, vários programadores relutam em realizar um cálculo manual assim que um programa realiza qualquer operação algébrica. A fobia matemática vem à tona e, irracionalmente, eles esperam poder evitar a álgebra e sobrepujar o programa por meio de uma reformulação aleatória, por exemplo, reorganizando os sinais `+` e `-`. Reformulação aleatória é sempre um grande desperdício de tempo e raramente leva a resultados úteis.

Vamos examinar outra classe `TaxReturn`. Suponha que um contribuinte solteiro ganhe US$ 50.000. As regras na Tabela 1 determinam que os primeiros US$ 21.450 são taxados em 15%. Eventualmente você terá de usar uma calculadora – números do mundo real normalmente são detestáveis. Calcule 21.450 x 0,15 = 3.217,50. Em seguida, como US$ 50.000 é menor que o maior limite da segunda faixa, todos os valores acima de US$ 21.450, são taxados em 28%. Isto é (50.000-21.450) x 0,28 = 7.994. O imposto total é a soma, 3.217,50 + 7.994 = 11.211,50. Isso não foi tão difícil.

Execute o programa e compare os resultados. Como os resultados batem, ganhamos mais confiança quanto à exatidão do programa.

É ainda melhor fazer cálculos manuais antes de escrever o programa. Fazer isso ajuda a entender a tarefa à mão e você será capaz de implementar sua solução mais rapidamente.

Dica de qualidade 5.4

Prepare os casos de teste antecipadamente

Vamos considerar como podemos testar o programa de cálculo de imposto. Naturalmente, não podemos testar todas as possíveis entradas do tipo de declaração e do nível de renda. Mesmo se pudéssemos, não haveria razão em testar todas. Se o programa calcular corretamente um ou dois valores de imposto em uma dada faixa, teremos então uma boa razão para acreditar que todos os valores dentro dessa faixa estão corretos. Nosso objetivo é a cobertura completa de todos os casos.

Há duas possibilidades para o tipo de declaração e três faixas de imposto para cada tipo. Isso exige seis casos de teste. Queremos então testar *condições de erro,* como uma renda negativa. Isso cria sete casos de teste. Para os seis primeiros, precisamos calcular manualmente qual resposta esperamos. Para o último, precisamos saber quais informes de erro são esperados. Anotamos os casos de teste e então começamos a codificar.

Você realmente deve testar sete entradas para esse programa simples? Certamente sim. Além disso, se encontrar um erro no programa que não foi coberto por um dos casos de teste, crie outro caso de teste e o adicione à sua coleção. Depois de corrigir os erros conhecidos, *execute todos os casos de teste novamente.* A experiência mostra que os casos que você acabou de tentar corrigir provavelmente estão funcionando agora, mas aqueles erros que você corrigiu há duas ou três iterações têm uma boa chance de reaparecer! Se descobrir que um erro reaparece várias vezes, normalmente é um bom sinal de que você não entendeu completamente alguma parte da interação sutil entre os recursos do seu programa.

Sempre é uma boa idéia projetar casos de teste *antes de* começar a codificar. Há duas razões para isso. A análise dos casos de teste fornece um melhor entendimento do algoritmo que você está em vias de programar. Além disso, já foi observado que os programadores instintivamente não gostam de testar as partes frágeis do código. É difícil acreditar, mas, freqüentemente, você fará essa observação no seu próprio trabalho. Preste atenção quando uma outra pessoa testa seu programa. Haverá ocasiões em que ela fará uma entrada que o deixará muito nervoso porque você não se sente seguro de que seu programa poderá tratá-la e você nunca ousou testá-la. Esse é um fenômeno bem-conhecido e criar o plano de teste antes de escrever o código oferece alguma proteção.

Tópico avançado 5.6

Registro em log

Às vezes você executa um programa e não sabe onde ele gasta seu tempo. Para obter uma listagem do fluxo do programa, você pode inserir mensagens de rastreamento, como esta:

```
public double getTax()
{
   . . .
   if (status == SINGLE)
   {
      System.out.println("status is SINGLE");
      . . .
   }
   . . .
}
```

Mas há um problema com o uso do `System.out.println` para mensagens de rastreamento. Depois de testar o programa, você precisa remover todas as instruções de impressão que criam as mensagens de rastreamento. Se, porém, encontrar um outro erro, você precisará reativar as instruções de impressão.

Para solucionar esse problema, utilize a classe `Logger`, que permite desativar as mensagens de rastreamento sem removê-las do programa.

Em vez de imprimir `System.out` diretamente, utilize o objeto de logger global `Logger.global` e chame

```
Logger.global.info("status is SINGLE");
```

Mensagens de log podem ser desativadas quando o teste está completo.

Por padrão, a mensagem é impressa. Mas se você chamar:

```
Logger.global.setLevel(Level.OFF);
```

no começo do método `main` do seu programa, todas as mensagens de log serão suprimidas. Portanto, você pode desativar as mensagens de log quando seu programa funciona bem e reativá-las se encontrar um outro erro. Em outras palavras, utilizar `Logger.global.info` é o mesmo que `System.out.println`, exceto que você pode ativar e desativar facilmente as mensagens de log.

Um truque comum para rastrear o fluxo de execução é criar mensagens de log para quando um método é chamado e para quando ele retorna. No começo de um método, imprima os parâmetros:

```
public TaxReturn(double anIncome, int aStatus)
{
   Logger.global.info("Parameters: anIncome = " + anIncome
         + " aStatus = " + aStatus);
   . . .
}
```

No final de um método, imprima o valor de retorno:

```
public double getTax()
{
   . . .
   Logger.global.info("Return value = " + tax);
   return tax;
}
```

A classe `Logger` tem muitas outras opções para mensagens de log de capacidade industrial. Confira a documentação da API se quiser ter mais controle sobre o registro em log.

RESUMO DO CAPÍTULO

1. A instrução `if` deixa um programa executar diferentes ações com base em uma condição.
2. Uma instrução de bloco agrupa várias instruções.
3. Operadores relacionais comparam valores. O operador `==` testa a igualdade.
4. Ao comparar números de ponto flutuante, não teste a igualdade. Em vez disso, verifique se eles são suficientemente próximos.
5. Não utilize o operador `==` para comparar strings. Em vez disso, utilize o método `equals`.
6. O método `compareTo` compara strings na ordem do dicionário.
7. O operador `==` testa se duas referências de objeto são idênticas. Para comparar o conteúdo dos objetos, você precisa utilizar o método `equals`.
8. A referência `null` não referencia nenhum objeto.
9. Múltiplas condições podem ser combinadas para avaliar decisões complexas. O arranjo correto depende da lógica do problema a ser solucionado.
10. O tipo `boolean` tem dois valores: `true` e `false`.
11. Um método de predicado retorna um valor `boolean`.
12. Você pode desenvolver testes complexos com os operadores booleanos `&&` (e), `||` (ou) e `!` (não).
13. Você pode armazenar o resultado de uma condição em uma variável booleana.
14. O teste da caixa preta descreve um método de teste que não leva em consideração a estrutura da implementação.
15. O teste da caixa preta usa as informações sobre a estrutura de um programa.
16. A cobertura do teste é uma medida de quantas partes de um programa foram testadas.
17. Casos de teste de limites são os casos de teste que estão no limite das entradas aceitáveis.
18. Você deve calcular casos de teste manualmente para garantir que seu aplicativo calcule a resposta correta.
19. Mensagens de log podem ser desativadas quando o teste está completo.

LEITURA ADICIONAL

1. `http://www.irs.ustreas.gov` Site do Internal Revenue Service.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.lang.Character
   isDigit
   isLetter
   isLowerCase
   isUpperCase
   java.lang.Object
   equals
java.lang.String
   equalsIgnoreCase
   compareTo
java.util.logging.Level
   ALL
   INFO
   NONE
java.util.logging.Logger
   getLogger
   info
   setLevel
java.util.Scanner
   hasNextDouble
   hasNextInt
```

EXERCÍCIOS DE REVISÃO

★★ **Exercício R5.1.** Encontre os erros nas instruções `if` a seguir.

a.
```
if quarters > 0 then System.out.println(quarters + " quarters");
```

b.
```
if (1 + x > Math.pow(x, Math.sqrt(2)) y = y + x;
```

c.
```
if (x = 1) y++; else if (x = 2) y = y + 2;
```

d.
```
if (x && y == 0) { x = 1; y = 1; }
```

e.
```
if (1 <= x <= 10)
   System.out.println(x);
```

f.
```
if (!s.equals("nickels") || !s.equals("pennies")
      || !s.equals("dimes") || !s.equals("quarters"))
   System.out.print("Input error!");
```

g.
```
if (input.equalsIgnoreCase("N") || "NO")
   return;
```

h.
```
int x = Integer.parseInt(input);
if (x != null) y = y + x;
```

i.
```
language = "English";
if (country.equals("US"))
   if (state.equals("PR")) language = "Spanish";
else if (country.equals("China"))
   language = "Chinese";
```

★ **Exercício R5.2.** Explique os termos abaixo e dê um exemplo para cada construção:

a. Expressão
b. Condição
c. Instrução
d. Instrução simples
e. Instrução composta
f. Bloco

★ **Exercício R5.3.** Explique a diferença entre uma instrução `if/else if/else` e instruções `if` aninhadas. Dê um exemplo de cada uma.

★ **Exercício R5.4.** Forneça um exemplo de uma instrução `if/else` `if/else` na qual a ordem dos testes não é importante. Dê um exemplo em que a ordem dos testes é importante.

★ **Exercício R5.5.** Entre os pares de strings a seguir, qual vem primeiro na ordenação lexicográfica?

a. `"Tom", "Dick"`
b. `"Tom", "Tomato"`
c. `"church", "Churchill"`
d. `"car manufacturer", "carburetor"`
e. `"Harry", "hairy"`
f. `"C++", " Car"`
g. `"Tom", "Tom"`
h. `"Car", "Carl"`
i. `"car", "bar"`
j. `"101", "11"`
k. `"1.01", "10.1"`

★ **Exercício R5.6.** Complete a tabela verdade a seguir encontrando os valores reais das expressões booleanas para todas as combinações de entradas booleanas `p`, `q` e `r`.

`p`	`q`	`r`	`(p && q) || !r`	`!(p && (q || !r))`
falso	falso	falso		
falso	falso	verdadeiro		
falso	verdadeiro	falso		
. . .				
mais 5 combinações				
. . .				

★ **Exercício R5.7.** Antes de você implementar um algoritmo complexo, uma boa idéia é entendê-lo e analisá-lo. O objetivo deste exercício é ganhar um melhor entendimento do algoritmo de cálculo de imposto da Seção 5.3.2.

Um recurso do código de imposto é a *penalidade do casamento*. Sob certas circunstâncias, um casal paga impostos mais altos do que a soma que os dois parceiros pagariam se fossem solteiros. Descubra exemplos desses níveis de renda.

★★★ **Exercício R5.8.** Verdadeiro ou falso? *A* `&&` *B* é o mesmo que *B* `&&` *A* para qualquer condição booleana *A* e *B*.

★ **Exercício R5.9.** Explique a diferença entre

```
s = 0;
if (x > 0) s++;
if (y > 0) s++;
```

e

```
s = 0;
if (x > 0) s++;
else if (y > 0) s++;
```

★★ **Exercício R5.10.** Utilize a lei de De Morgan para simplificar estas expressões booleanas.

a. `!(x > 0 && y > 0)`

b. `!(x != 0 || y != 0)`

c.
```
!(country.equals("US") && !state.equals("HI")
        && !state.equals("AK"))
```

d. `!(x % 4 != 0 || !(x % 100 == 0 && x % 400 == 0))`

★★ **Exercício R5.11.** Crie um outro exemplo de código Java que mostra o problema do `else` oscilante, utilizando a instrução a seguir: Um aluno com uma média de notas de pelo menos 1.5, mas menor que 2, está em recuperação; com menos de 1.5, o aluno é reprovado.

★ **Exercício R5.12.** Explique a diferença entre o operador `==` e o método `equals` ao comparar strings.

★★ **Exercício R5.13.** Explique a diferença entre os testes

```
r == s
```

e

```
r.equals(s)
```

onde tanto `r` como `s` são do tipo `Rectangle`.

★★★ **Exercício R5.14.** O que há de errado com este teste para verificar se `r` é `null`? O que acontece quando este código é executado?

```
Rectangle r;
. . .
if (r.equals(null))
   r = new Rectangle(5, 10, 20, 30);
```

★ **Exercício R5.15.** Explique como a ordenação lexicográfica de strings difere do ordenamento das palavras em um dicionário ou em uma lista telefônica. *Dica:* Considere strings, como `IBM`, `wiley.com`, `Century 21`, `While-U-Wait` e `7-11`.

★★★ **Exercício R5.16.** Escreva o código Java para testar se dois objetos do tipo `Line2D.Double` representam a mesma linha quando exibidos na tela gráfica. *Não* utilize `a.equals(b)`.

```
Line2D.Double a;
Line2D.Double b;

if (sua condição entra aqui)
   g2.drawString("They look the same!", x, y);
```

Dica: Se `p` e `q` forem pontos, então `Line2D.Double(p, q)` e `Line2D.Double(q, p)` parecerão idênticos.

★ **Exercício R5.17.** Explique por que é mais difícil comparar números de ponto flutuante do que inteiros. Escreva o código Java para testar se um inteiro `n` é igual a 10 e se um número de ponto flutuante `x` é igual a 10.

★★ **Exercício R5.18.** Considere o teste a seguir para ver se um ponto está dentro de um retângulo.

```
Point2D.Double p = . . .
Rectangle r = . . .
boolean xInside = false;
if (r.getX() <= p.getX() && p.getX() <= r.getX() + r.getWidth())
   xInside = true;
boolean yInside = false;
if (r.getY() <= p.getY() && p.getY() <= r.getY() + r.getHeight())
   yInside = true;
if (xInside && yInside)
   g2.drawString("p is inside the rectangle.",
         p.getX(), p.getY());
```

Reescreva esse código para eliminar os valores `true` e `false` explícitos, configurando `xInside` e `yInside` como os valores das expressões booleanas.

★T **Exercício R5.19.** Forneça um conjunto de casos de teste para o programa de terremoto na Seção 5.3.1. Assegure a cobertura de todos os desvios.

★★T **Exercício R5.20.** Forneça um conjunto de casos de teste para o programa de imposto na Seção 5.3.2. Calcule os resultados esperados manualmente.

★T **Exercício R5.21.** Dê um exemplo de um caso de teste de limites para o programa de imposto na Seção 5.3.2. Qual resultado você espera?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★★ **Exercício P5.1.** Escreva um programa que imprime todas as soluções reais para a equação quadrática $ax^2 + bx + c = 0$. Interprete a, b, c e utilize a fórmula quadrática. Se o *discriminante* $b^2 - 4aC$ for negativo, exiba uma mensagem que declare que não há nenhuma solução real.

Implemente uma classe `QuadraticEquation` cujo construtor recebe os coeficientes `a`, `b`, `c` da equação quadrática. Forneça métodos `getSolution1` e `getSolution2` que obtêm as soluções, utilizando a fórmula quadrática ou 0 se não houver solução. O método `getSolution1` deve retornar a menor das duas soluções.

Forneça um método:

```
boolean hasSolutions()
```

que retorna `false` se o descriminante for negativo.

★★ **Exercício P5.2.** Escreva um programa que receba a entrada de usuário descrevendo uma carta de baralho na notação de abreviação a seguir:

Notação	Significado
A	
2 . . . 10	Valores das cartas
J	Valete
Q	Rainha
K	Rei
D	Ouros
H	Copas
S	Espadas
C	Paus

Seu programa deve imprimir a descrição completa da carta. Por exemplo,

```
Enter the card notation:
4S
Four of spades
```

Implemente uma classe `Card` cujo construtor recebe a string de notação da carta e cujo método `getDescription` retorna uma descrição da carta. Se a string de notação não estiver no formato correto, o método `getDescription` deverá retornar a string `"Unknown"`.

★★ **Exercício P5.3.** Escreva um programa que lê três números de ponto flutuante e imprime as três entradas na ordem de classificação. Por exemplo:

```
Please enter three numbers:
4
9
2.5
The inputs in sorted order are:
2.5
4
9
```

★ **Exercício P5.4.** Escreva um programa que imprime a pergunta "Do you want to continue?" ["Você quer continuar?"] e lê uma entrada de usuário. Se a entrada de usuário for "Y", "Yes", "OK", "Sure" ou "Why not?", imprima "OK". Se a entrada de usuário for "N" ou "No", imprima então "Terminating". Caso contrário, imprima "Bad input" [Entrada inválida]. A entrada do usuário não deve ser importante. Por exemplo, "y" ou "yes" também são entradas válidas. Escreva uma classe `YesNoChecker` para esse propósito.

★ **Exercício P5.5.** Escreva um programa que converte uma avaliação escolar baseada em letras para uma nota numérica. As letras das notas são `A B C D F`, possivelmente seguidas por `+` ou `-`. Seus valores numéricos são 4, 3, 2, 1 e 0. Não há `F+` ou `F-`. Um `+` aumenta o valor numérico em 0,3, um `-` diminui o valor em 0,3. Mas um `A+` tem o valor 4,0.

```
Enter a letter grade:
B-
Numeric value: 2.7.
```

Utilize uma classe `Grade` com um método `getNumericGrade`.

★ **Exercício P5.6.** Escreva um programa que traduz um número na letra mais próxima. Por exemplo, o número 2.8 (que poderia ter sido a média de várias notas) seria convertido em `B-`. A conversão ocorre em relação à melhor nota; por exemplo, 2.85 deve ser um `B`.

Utilize uma classe `Grade` com um método `getLetterGrade`.

★ **Exercício P5.7.** Escreva um programa que lê três strings e imprime a menor e a maior lexicograficamente:

```
Please enter three strings:
Tom
Dick
Harry
The inputs in sorted order are:
Dick
Harry
Tom
```

★★ **Exercício P5.8.** Altere a implementação do método `getTax` na classe `TaxReturn`, configurando variáveis `bracket1` e `bracket2`, dependendo do estado civil. Crie então uma única fórmula que calcule o imposto, dependendo da renda e das faixas. Verifique se seus resultados são idênticos àqueles da classe `TaxReturn` neste capítulo.

★ **Exercício P5.9.** Um ano com 366 dias é chamado *ano bissexto*. Um ano é um ano bissexto se for divisível por 4 (por exemplo, 1980). Mas desde a introdução do calendário gregoriano em 15 de outubro de 1582, um ano não é um ano bissexto se ele for divisível por 100 (por exemplo, 1900); entretanto, será um ano bissexto se for divisível por 400 (por exemplo, 2000). Escreva um programa que solicita ao usuário um ano e calcula se é um ano bissexto. Implemente uma classe `Year` com um método de predicado `boolean isLeapYear()`.

★ **Exercício P5.10.** Escreva um programa que solicita ao usuário inserir um mês (1 = janeiro, 2 = fevereiro e assim por diante) e então imprime o número de dias do mês. Para fevereiro, imprime "28 dias".

```
Enter a month (1-12):
5
31 days
```

Implemente uma classe `Month` com um método `int getDays()`.

★★★ **Exercício P5.11.** Escreva um programa que lê dois números de ponto flutuante e (a) testa se eles são os mesmos quando arredondados para duas casas decimais e (b) se diferem por menos de 0.01. Eis duas execuções de exemplo.

```
Enter two floating-point numbers:
2.0
1.99998
They are the same when rounded to two decimal places.
They differ by less than 0.01.

Enter two floating-point numbers:
0.999
0.991
They are different when rounded to two decimal places.
They differ by less than 0.01.
```

★ **Exercício P5.12.** Melhore a classe `BankAccount` do Capítulo 3

- Rejeitando quantidades negativas nos métodos `deposit` e `withdraw`
- Rejeitando saques que resultariam em um saldo negativo

★ **Exercício P5.13.** Escreva um programa que lê o salário por hora de um funcionário. Então solicite quantas horas o funcionário trabalhou na última semana. Certifique-se de aceitar horas fracionárias. Calcule o salário. Todas as horas extras (acima de 40 horas por semana) são pagas em 150% do salário normal. Resolva esse problema implementando uma classe `Paycheck`.

★★ **Exercício P5.14.** Escreva um programa de conversão de medidas que solicita que usuários identifiquem uma medida da qual eles querem converter e uma medida para a qual converter. Medidas válidas são *in*, *ft*, *mi*, *mm*, *cm*, *m* e *km*. Defina dois objetos de uma classe `UnitConverter`, que converte entre metros e uma dada medida.

```
Convert from:
in
Convert to:
mm
Value:
10
10 in = 254 mm
```

★★★ **Exercício P5.15.** Uma linha no plano pode ser especificada de várias maneiras:

- fornecendo um ponto (x, y) e uma declividade m
- fornecendo dois pontos $(x1, y1)$, $(x2, y2)$
- como uma equação na forma de interceptação da declividade $y = mx + b$
- como uma equação $x = a$ se a linha for vertical

Implemente uma classe `Line` com quatro construtores que correspondam aos quatro casos anteriores. Implemente métodos

```
boolean intersects(Line other)
boolean equals(Line other)
boolean isParallel(Line other)
```

★★G **Exercício P5.16.** Escreva um programa que desenha um círculo com raio 100 e centro (200, 200). Solicite que o usuário especifique as coordenadas x e y de um ponto. Desenhe o ponto como um pequeno círculo. Se o ponto residir dentro do círculo, torne o pequeno círculo verde. Caso contrário, torne-o vermelho. No seu exercício, defina uma classe `Circle` e um método `boolean isInside(Point2D.Double p)`.

★★★G **Exercício P5.17.** Escreva um programa gráfico que solicita que o usuário especifique os raios dos dois círculos. O primeiro círculo tem centro (100, 200) e o segundo círculo tem centro (200, 100). Desenhe os círculos. Se eles se interseccionarem, torne os dois círculos verdes. Caso contrário, torne-os vermelhos. *Dica:* Calcule a distância entre os centros e compare com os raios. Seu programa não deve desenhar nada se o usuário inserir um raio negativo. No seu exercício, defina uma classe `Circle` e um método `boolean intersects(Circle other)`.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 5.1.** Implemente uma classe de *bloqueio de combinação*. Um bloqueio de combinação tem um seletor com 26 posições rotuladas A. . . Z. O seletor precisa ser configurado três vezes. Se ele for configurado com a combinação correta, o bloqueio pode ser aberto. Quando o bloqueio é fechado novamente, a combinação pode ser inserida outra vez. Se um usuário configurar o seletor mais de três vezes, as três últimas configurações determinam se o bloqueio pode ser aberto. Uma parte importante deste exercício é implementar uma interface adequada para a classe `CombinationLock`.

★★★ **Projeto 5.2.** Acesse as instruções do formulário 1040 do último ano em `http://www.irs.ustreas.gov` [1]. Descubra as faixas de imposto utilizadas no último ano para todas as categorias de contribuintes (solteiro, declaração conjunta de casados, declaração separada de casados e chefe de família). Escreva um programa que calcule os impostos seguindo essa declaração. Ignore deduções, isenções e créditos. Simplesmente aplique a alíquota de imposto à renda.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

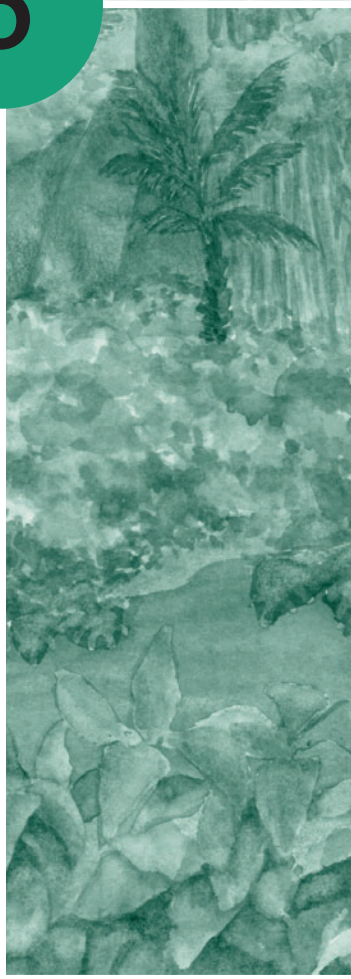
1. Se o valor de saque for igual ao saldo, o resultado deverá ser um saldo zero e nenhuma multa.
2. Somente a primeira instrução de atribuição é parte da instrução `if`. Utilize chaves para agrupar ambas as instruções de atribuição em uma instrução de bloco.
3. (a) 0; (b) 1; (c) uma exceção é lançada
4. Sintaticamente incorreta: e, g, h. Logicamente questionável: a, f, d
5. Sim, se você também inverter as comparações:

```
if (richter < 3.5)
   r = "Generally not felt by people";
else if (richter < 4.5)
   r = "Felt by many people, no destruction";
else if (richter < 6.0)
   r = "Damage to poorly constructed buildings";
. . .
```

6. A faixa mais alta de imposto só é aplicada sobre a renda na faixa mais alta. Suponha que você seja solteiro e tenha um rendimento de US$ 51.800. Você deveria tentar obter um aumento salarial de US$ 200? Absolutamente – você precisa manter 72% do primeiro US$ 100 e 69% do próximo $100.
7. Quando `x` é zero.
8. `if (!Character.isDigit(ch)) . . .`
9. 7
10. Uma entrada de 0 deve gerar uma saída de `"Generally not felt by people"` [Em geral, não sentido pelas pessoas]. (Se a saída for `"Negative numbers are not allowed"` [Números negativos não são permitidos], haverá um erro no programa.

Capítulo 6

Iteração



Objetivos do capítulo

- Ser capaz de programar laços com as instruções `while` e `for`
- Evitar laços infinitos e erros "por um"
- Entender laços aninhados
- Aprender a processar entrada
- Implementar simulações

T Aprender sobre o depurador

Este capítulo apresenta as várias construções de iteração da linguagem Java. Essas construções executam uma ou mais instruções repetidamente até que um objetivo seja alcançado. Veremos como as técnicas que você aprende neste capítulo podem ser aplicadas ao processamento dos dados de entrada e à programação de simulações.

Conteúdo do Capítulo

6.1 Laços `while` 226
Sintaxe 6.1: O comando instrução `while` 228
Erro comum 6.1: Laços infinitos 229
Erro comum 6.2: Erros "por um" 229
Tópico avançado 6.1: Laços do ⊕
Fato aleatório 6.1: Código espaguete ⊕

6.2 Laços `for` 231
Sintaxe 6.2: O comando instrução `for` 231
Dica de qualidade 6.1: Utilize o laço `for` apenas para os propósitos para os quais ele foi projetado 235
Erro comum 6.3: Esquecendo um ponto-e-vírgula 236
Erro comum 6.4: Um ponto-e-vírgula em excesso 236
Dica de qualidade 6.2: Não utilize `!=` para testar o fim de um intervalo 237
Tópico avançado 6.2: Variáveis definidas no cabeçalho de um laço `for` ⊕

6.3 Laços aninhados 237

6.4 Processando valores sentinela 238
Como fazer 6.1: Implementando laços 242
Dica de qualidade 6.3: Limites simétricos e assimétricos 244
Dica de qualidade 6.4: Conte as iterações 244
Tópico avançado 6.3: O problema do "um laço e meio" ⊕
Tópico avançado 6.4: Os comandos `break` e `continue` ⊕

6.5 Números aleatórios e simulações 245
Tópico avançado 6.5: Invariantes de laço ⊕
Fato aleatório 6.2: Provas de correção ⊕

6.6T Utilizando um depurador 248

6.7T Uma sessão de depuração de exemplo 251
Como fazer 6.2: Depurando 257
Fato aleatório 6.3: O primeiro bug ⊕

6.1 Laços `while`

Neste capítulo você aprenderá a escrever programas que executam repetidamente uma ou mais instruções. Ilustraremos esses conceitos examinando situações de um investimento típico. Considere uma conta bancária com um saldo inicial de US$ 10.000 que rende 5% de juros. Os juros são calculados no final de cada ano sobre o saldo atual e então depositados na conta bancária. Por exemplo, depois do primeiro ano, a conta rendeu US$ 500 (5% de $10.000) de juros. Os juros são adicionados à conta bancária. No ano seguinte, os juros correspondem a US$ 525 (5% de US$ 10.500) e o saldo é de US$ 11.025. A Tabela 1 mostra como o saldo aumenta nos primeiros cinco anos.

Em quantos anos o saldo alcançará US$ 20.000? Naturalmente, ele não levará mais de 20 anos, porque pelo menos US$ 500 são adicionados à conta bancária cada ano. Mas levará menos de 20 anos, porque os juros são calculados sobre saldos cada vez maiores. Para conhecer a resposta exata, escreveremos um programa que adiciona juros repetidamente até o saldo ser alcançado.

Uma instrução `while` executa um bloco de código repetidamente. Uma condição controla a freqüência com que o laço é executado.

Em Java, a instrução `while` implementa essa repetição. A construção:

```
while (condição)
   instrução
```

continua executando a instrução enquanto a condição for verdadeira.

Tabela 1 Rendimento de um investimento

Ano	Saldo
0	$10.000,00
1	$10.500,00
2	$11.025,00
3	$11.576,25
4	$12.155,06
5	$12.762,82

Na maioria das vezes, a instrução é uma instrução de bloco, isto é, um conjunto de instruções delimitado por `{ }`.

No nosso caso, queremos saber quando a conta bancária alcançou um saldo particular. Enquanto o saldo for menor, continuaremos a adicionar juros e a incrementar o contador `year`:

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Para o texto integral do programa de exemplo que resolve nosso problema de investimento, consulte `ch06/invest1/` no código-fonte, no material online do livro, disponível no site www.bookman.com.br.

Uma instrução `while` é freqüentemente chamada *loop* (laço). Se desenhar um fluxograma, você verá que o controle faz um loop para trás em relação ao teste depois de cada iteração (veja Figura 1).

O laço a seguir,

```
while (true)
   instrução
```

executa a instrução repetidamente, sem jamais terminar. Opa, espere aí! Por que você faria isso? O programa nunca pararia. Há duas razões. Na realidade alguns programas nunca param; o software que controla um caixa automático, uma central telefônica ou um forno de microondas nunca pára (pelo menos não até que o dispositivo seja desligado). Nossos programas normalmente não são desse tipo, mas mesmo que não seja possível terminar o laço, você pode fazê-lo sair do método que o contém. Isso pode ser útil quando o teste de conclusão cai naturalmente no meio do laço (veja o Tópico Avançado 6.3).

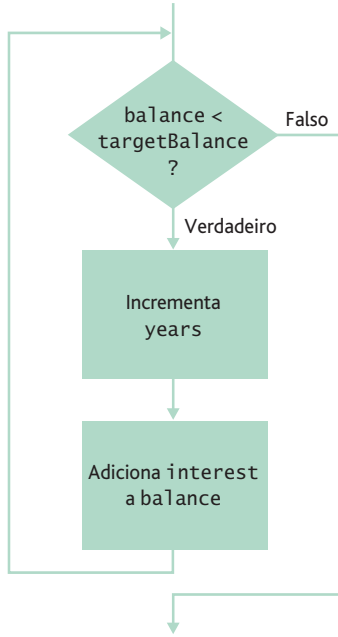


Figura 1
Fluxograma de um laço `while`.

SINTAXE 6.1 O comando instrução `while`

```
while (condição)
   instrução
```

Exemplo:

```
while (balance < targetBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Objetivo:

Executar repetidamente uma instrução enquanto uma condição for verdadeira

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Com que freqüência a instrução a seguir é executada no laço?
   `while (false)` *instrução*;
2. O que aconteceria se `RATE` fosse configurado como 0 no método `main` do programa `InvestmentRunner`?

ERRO COMUM 6.1

Laços infinitos

O erro de laço que mais incomoda é o laço infinito: um laço que executa eternamente e que só pode ser parado matando o programa à força ou reiniciando o computador. Se houver instruções de saída no laço, quantidades cada vez maiores da saída serão exibidas na tela, fazendo-a piscar. Caso contrário, o programa ficará aparentemente travado, parecendo não fazer nada. Em alguns sistemas você pode finalizar à força um programa travado pressionando Ctrl+Break ou Ctrl+C. Em outros, você pode fechar a janela em que o programa está em execução.

Uma razão comum para os laços infinitos é esquecer de avançar a variável que controla o laço:

```
int years = 0;
while (years < 20)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Aqui o programador se esqueceu de adicionar uma instrução para incrementar `years` no laço. Como resultado, o valor de `years` sempre permanece `0` e o laço nunca pára.

Uma outra razão comum para um laço infinito é incrementar acidentalmente um contador que deveria ser decrementado (ou vice-versa). Considere este exemplo:

```
int years = 20;
while (years > 0)
{
   years++; // Ops, deveria ser years--
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Na verdade, a variável `years` deveria ter sido decrementada, não incrementada. Esse é um erro comum, porque incrementar contadores é muito mais comum que decrementá-los e é por isso que você digita `++` quase que automaticamente. Como conseqüência, `years` sempre é maior que `0` e o laço nunca termina. (Na realidade, `years` em algum momento excederá o maior inteiro positivo representável e utilizará um número negativo. Então o laço pára – naturalmente, isso leva bastante tempo e o resultado é completamente errado.)

ERRO COMUM 6.2

Erros "por um"

Considere nosso cálculo do número de anos necessário para dobrar um investimento:

```
int years = 0;
while (balance < 2 * initialBalance)
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
```

```
   }
   System.out.println(
         "The investment reached the target after "
         + years + " years.");
```

`years` deveria iniciar em 0 ou em 1? Você deveria testar `balance < 2 * initialBalance` ou `balance <= 2 * initialBalance`? É fácil haver um erro "por um" nessas expressões.

Algumas pessoas tentam resolver erros "por um" inserindo aleatoriamente +1 ou -1 até que o programa pareça funcionar. Isso é, naturalmente, uma estratégia horrível. Pode levar muito tempo para compilar e testar todas as várias possibilidades. Um pequeno esforço mental pode nos economizar bastante tempo.

> Um erro "por um" é um erro comum ao programar laços. Pense por meio de casos de teste simples para evitar esse tipo de erro.

Felizmente, erros "por um" são fáceis de evitar, simplesmente pense por meio de alguns casos de teste e utilize as informações a partir desses casos para chegar a um fundamento lógico para a condição correta do laço.

`years` deveria iniciar em 0 ou em 1? Examine um cenário com valores simples: um saldo inicial de US$ 100 e uma taxa de juros de 50%. Depois do ano 1, o saldo é US$ 150 e depois do ano 2 ele é US$ 225 ou mais de US$ 200. Portanto, o investimento dobrou depois de 2 anos. O laço executou duas vezes, incrementando `years` a cada vez. Conseqüentemente `years` deve iniciar em 0, não em 1.

Em outras palavras, a variável `balance` indica o saldo *depois* do fim do ano. No início, a variável `balance` contém o saldo depois do ano 0 e não depois do ano 1.

Em seguida, você deveria utilizar uma comparação < ou <= no teste? Isso é mais difícil de descobrir, porque é raro o saldo ser exatamente duas vezes o saldo inicial. Naturalmente, há um caso em que isso acontece, a saber, quando os juros são 100%. O laço executa uma vez. Agora `years` é 1 e `balance` é exatamente igual a `2 * initialBalance`. O investimento dobrou depois de um ano? Sim. Portanto, o laço *não* deve executar novamente. Se a condição de teste for `balance < 2 * initialBalance`, o laço pára, como deveria. Se a condição de teste fosse `balance <= 2 * initialBalance`, o laço seria executado mais uma vez.

Em outras palavras, você continua a adicionar juros enquanto o saldo *ainda não dobrou*.

Tópico avançado 6.1

Laços do

O Tópico Avançado 6.1 discute o laço `do`, uma construção opcional de laço que testa a condição do laço no fim do seu corpo.

Fato aleatório 6.1

Código espaguete

O Fato Aleatório 6.1 discute o comando "`goto`" que fazia parte de várias linguagens de programação mais antigas. O comando "`goto`" pode levar a um fluxo de controle complexo, ridiculamente chamado de "código espaguete".

6.2 Laços for

Um dos tipos mais comuns de laço tem a forma:

```
i = início;
while (i <= fim)
{
   . . .
   i++;
}
```

Como esse laço é tão comum, há uma forma especial dele que enfatiza o padrão:

```
for (i = início; i <= fim; i++)
{
   . . .
}
```

Você também pode declarar a variável contadora do laço dentro do cabeçalho do laço `for`. Essa abreviação conveniente restringe o uso da variável ao corpo do laço (como será discutido detalhadamente no Tópico Avançado 6.2).

```
for (int i = início; i <= fim; i++)
{
   . . .
}
```

Vamos utilizar esse laço para descobrir o tamanho do nosso investimento de US$ 10.000 se 5% de juros forem acumulados por 20 anos. Naturalmente, o saldo será maior que US$ 20.000, porque pelo menos US$ 500 são adicionados todos os anos. Talvez você se surpreenda ao descobrir o quanto o saldo é maior.

SINTAXE 6.2 O comando instrução for

```
for (inicialização; condição; atualização)
   instrução
```

Exemplo:

```
for (i = 1; i <= n; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

Objetivo:

Executar uma inicialização e então continuar executando e atualizando uma expressão enquanto uma condição for verdadeira

Utilize um laço `for` quando uma variável for de um valor inicial a um valor final com um incremento ou decremento constante.

No nosso laço, deixamos que `i` vá de `1` a `n`, o número de anos pelos quais queremos acumular juros.

```
for (int i = 1; i <= n; i++)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

A Figura 2 mostra o fluxograma correspondente.

Os três espaços no cabeçalho `for` podem conter três expressões quaisquer. Você pode contar de baixo para cima em vez de cima para baixo:

```
for (years = n; years > 0; years--)
```

O incremento ou decremento não precisa ser em passos de 1:

```
for (x = -10; x <= 10; x = x + 0.5) . . .
```

É possível – embora seja um sinal de incrível mau gosto – adicionar condições não-relacionadas ao cabeçalho do laço:

```
for (rate = 5; years-- > 0; System.out.println(balance))
   . . . // Mau gosto
```

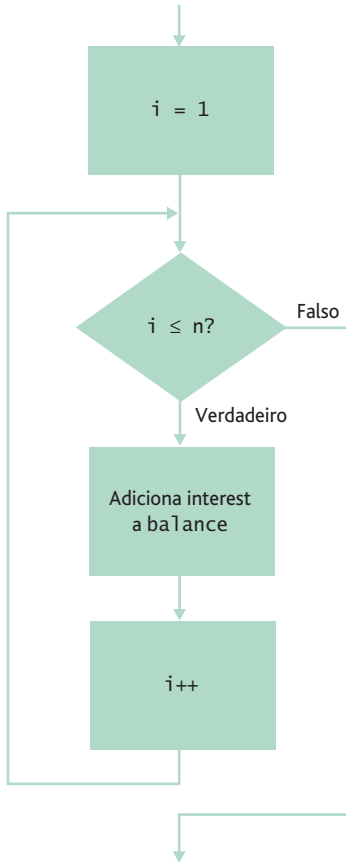


Figura 2 Fluxograma de um laço `for`.

Não vamos nem mesmo tentar decifrar o que ele poderia significar. Você só deve utilizar laços `for` que inicializam, testam e atualizam uma única variável.

ch06/invest2/Investment.java

```
/**
   Uma classe para monitorar o rendimento de um investimento que
   acumula juros a uma taxa anual fixa.
*/
public class Investment
{
   /**
      Constrói um objeto Investment a partir de um saldo inicial e uma
      taxa de juros.
      @param aBalance saldo inicial
      @param aRate taxa de juros em porcentagem
   */
   public Investment(double aBalance, double aRate)
   {
      balance = aBalance;
      rate = aRate;
      years = 0;
   }

   /**
      Continua a acumular juros até que um saldo alvo tenha
      sido alcançado.
      @param targetBalance saldo desejado
   */
   public void waitForBalance(double targetBalance)
   {
      while (balance < targetBalance)
      {
         years++;
         double interest = balance * rate / 100;
         balance = balance + interest;
      }
   }

   /**
      Continua a acumular juros por um dado número de anos.
      @param n número de anos
   */
   public void waitYears(int n)
   {
      for (int i = 1; i <= n; i++)
      {
         double interest = balance * rate / 100;
         balance = balance + interest;
   }
      years = years + n;
   }
```

```
48
49      /**
50         Obtém o saldo do investimento atual.
51         @return saldo atual
52      */
53      public double getBalance()
54      {
55         return balance;
56      }
57
58      /**
59         Obtém o número de anos que esse investimento acumulou
60         juros.
61         @return número de anos a partir do investimento inicial
62      */
63      public int getYears()
64      {
65         return years;
66      }
67
68      private double balance;
69      private double rate;
70      private int years;
71   }
```

ch06/invest2/InvestmentRunner.java

```
 1   /**
 2      Esse programa calcula o rendimento de um investimento em
 3      um dado número de anos.
 4   */
 5   public class InvestmentRunner
 6   {
 7      public static void main(String[] args)
 8      {
 9         final double INITIAL_BALANCE = 10000;
10         final double RATE = 5;
11         final int YEARS = 20;
12         Investment invest = new Investment(INITIAL_BALANCE, RATE);
13         invest.waitYears(YEARS);
14         double balance = invest.getBalance();
15         System.out.printf("The balance after %d years is %.2f\n",
16               YEARS, balance);
17      }
18   }
```

Saída

```
The balance after 20 years is 26532.98
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Reescreva o laço `for` no método `waitYears` como um laço `while`.
4. Quantas vezes o laço `for` a seguir é executado?

```
for (i = 0; i <= 10; i++)
   System.out.println(i * i);
```

DICA DE QUALIDADE 6.1

Utilize o laço for apenas para os propósitos para os quais ele foi projetado

Um laço `for` é uma *expressão idiomática* para uma forma particular de laço `while`. Um contador executa do início ao fim, com um incremento constante:

```
for (configura counter como start;testa se counter chegou a end;
      atualiza counter por increment)
{  . . .
   // counter, start, end, increment não são alterados aqui
}
```

Se seu laço não corresponder a esse padrão, não utilize a construção `for`. O compilador não impedirá que você escreva laços `for` inúteis:

```
// Estilo ruim – expressões de cabeçalho não relacionadas
for (System.out.println("Inputs:");
      (x = in.nextDouble()) > 0;
      sum = sum + x)
   count++;

for (int i = 1; i <= years; i++)
{
   // Estilo ruim – modifica o contador
   if (balance >= targetBalance)
      i = years + 1;
   else
   {
      double interest = balance * rate / 100;
      balance = balance + interest;
   }
}
```

Esses laços funcionarão, mas seu estilo é péssimo. Utilize um laço `while` para iterações que não se enquadram no padrão `for`.

ERRO COMUM 6.3

Esquecendo um ponto-e-vírgula

Ocasionalmente todo o trabalho de um laço é feito no cabeçalho do próprio laço. Suponha que você ignorou a Dica de Qualidade 6.1; você poderia então escrever um laço que dobra o investimento como a seguir:

```
for (years = 1;
      (balance = balance + balance * rate / 100) < targetBalance;
      years++)
   ;
System.out.println(years);
```

O corpo do laço `for` está completamente vazio, contendo apenas uma instrução vazia terminada por um ponto-e-vírgula.

Se encontrar um laço sem um corpo, é importante não esquecer do ponto-e-vírgula. Se o ponto-e-vírgula for omitido acidentalmente, a próxima linha se tornará parte da instrução do laço!

```
for (years = 1;
      (balance = balance + balance * rate / 100) < targetBalance;
      years++)
System.out.println(years);
```

Você pode evitar esse erro utilizando um bloco vazio `{ }` em vez de uma instrução vazia.

ERRO COMUM 6.4

Um ponto-e-vírgula em excesso

O que o laço a seguir imprime?

```
sum = 0;
for (i = 1; i <= 10; i++);
   sum = sum + i;
System.out.println(sum);
```

É claro que esse laço deveria calcular 1 + 2 + ... + 10 = 55. Mas, na verdade, a instrução print imprime 11!

Por que 11? Examine-o de novo. Você consegue identificar o ponto-e-vírgula no final do cabeçalho do laço `for`? Esse laço é um laço com um corpo vazio.

```
for (i = 1; i <= 10; i++)
   ;
```

O laço não faz nada 10 vezes e quando é terminado, `sum` ainda é 0 e `i` é 11. Em seguida, a instrução:

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

é executada e `sum` é 11. A instrução está recuada, o que engana o leitor humano. Mas o compilador não presta atenção a recuos.

Naturalmente, o ponto-e-vírgula no fim da instrução foi um erro de digitação. Alguém estava acostumado a digitar um ponto-e-vírgula no fim de cada linha e um ponto-e-vírgula foi adicionado acidentalmente ao laço `for`. O resultado é um laço com um corpo vazio.

DICA DE QUALIDADE 6.2

Não utilize `!=` para testar o fim de um intervalo

Eis um laço com um perigo oculto:

```
for (i = 1; i != n; i++)
```

O teste `i != n` é uma má idéia. Como o laço se comporta se `n` for zero ou negativo? O teste `i != n` nunca será falso, porque `i` inicia em 1 e aumenta a cada passo.

A correção é simples. Utilize `<=` em vez de `!=` na condição:

```
for (i = 1; i <= n; i++)
```

TÓPICO AVANÇADO 6.2

Variáveis definidas no cabeçalho de um laço `for`

O Tópico Avançado 6.2 mostra como definir múltiplas variáveis no cabeçalho de um laço `for` e explica que essas variáveis não são válidas além do corpo do laço.

6.3 Laços aninhados

Laços podem ser aninhados. Um exemplo típico de laços aninhados é imprimir uma tabela com linhas e colunas.

Às vezes, o corpo de um laço contém outro laço. Dizemos que o laço interno está *aninhado* dentro de um laço externo. Isso acontece freqüentemente quando você processa estruturas bidimensionais, como tabelas.

Vamos examinar um exemplo que parece um pouco mais interessante do que uma tabela de números. Queremos gerar a forma triangular a seguir:

```
[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
[][][][][][][]
```

A idéia básica é simples. Geramos uma seqüência de linhas:

```
for (int i = 1; i <= width; i++)
{
   // Cria uma linha do triângulo
   . . .
}
```

Como você cria uma linha de triângulo? Utilize outro laço para concatenar os quadrados `[]` para essa linha. Adicione então um caractere de nova linha ao final da linha. A `i`-ésima linha tem `i` símbolos, portanto o contador de laço vai de 1 até `i`.

```
for (int j = 1; j <= i; j++)
   r = r + "[]";
r = r + "\n";
```

Agrupar dois laços produz dois *laços aninhados:*

```
String r = "";
for (int i = 1; i <= width; i++)
{
   // Cria uma linha do triângulo
   for (int j = 1; j <= i; j++)
      r = r + "[]";
   r = r + "\n";
}
return r;
```

Para o texto integral do programa, consulte `ch06/triangle1/` no código-fonte disponível no material online do livro no site www.bookman.com.br.

Autoverificação da aprendizagem

5. Como você modificaria os laços aninhados para imprimir um quadrado em vez de um triângulo?
6. Qual é o valor de `n` depois dos seguintes laços aninhados?

```
int n = 0;
for (int i = 1; i <= 5; i++)
   for (int j = 0; j < i; j++)
      n = n + j;
```

6.4 Processando valores sentinela

Suponha que você queira processar um conjunto de valores, por exemplo, um conjunto de medidas. Seu objetivo é analisar os dados e exibir as propriedades do conjunto de dados, como o valor médio ou máximo. Você solicita ao usuário o primeiro valor, em seguida o segundo, então o terceiro e assim por diante. Quando a entrada termina?

Um método comum para indicar o fim de um conjunto de dados é um *valor sentinela*, um valor que não é parte dos dados. Em vez disso, o valor sentinela indica o término dos dados.

Alguns programadores escolhem números como 0 ou –1 como valores sentinela. Mas essa não é uma boa idéia. Esses valores podem muito bem ser entradas válidas. Uma idéia melhor é utilizar uma entrada que não seja um número, por exemplo, a letra `Q`. Eis a execução típica do programa:

```
Enter value, Q to quit: 1
Enter value, Q to quit: 2
Enter value, Q to quit: 3
Enter value, Q to quit: 4
Enter value, Q to quit: Q
Average = 2.5
Maximum = 4.0
```

Naturalmente, precisamos ler cada entrada como uma string, não como um número. Depois de testar se a entrada não é a letra `Q`, convertemos essa string em um número.

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
   Isso é tudo
else
{
   double x = Double.parseDouble(input);
   . . .
}
```

Às vezes, a condição de conclusão de um laço só pode ser avaliada no meio do laço. Você pode introduzir uma variável booleana para controlar esse tipo de laço.

Agora temos outro problema. O teste do término de laço ocorre no *meio* do laço, não na parte superior ou inferior. Primeiro você deve tentar ler a entrada antes de testar se alcançou o fim da entrada. Em Java, não há uma estrutura de controle pronta para o padrão "faça isso, teste e então faça uma outra coisa". Portanto, utilizamos uma combinação de um laço `while` e de uma variável `boolean`.

```
boolean done = false;
while (!done)
{
   Imprime o prompt
   String input = lê a entrada;
   if (fim da entrada indicada)
      done = true;
   else
   {
      Processa a entrada
   }
}
```

Esse padrão é, às vezes, chamado de "um laço e meio". Alguns programadores acham desajeitado introduzir uma variável de controle para um laço assim. O Tópico Avançado 6.3 mostra várias alternativas.

Vamos montar o programa de análise de dados. Para desassociar o tratamento da entrada do cálculo dos valores médios e máximos, introduziremos uma classe `DataSet`. Você adiciona valores a um objeto `DataSet` com o método `add`. O método `getAverage` retorna o valor médio de todos os dados adicionados e o método `getMaximum` retorna o maior valor.

ch06/dataset/DataAnalyzer.java

```
1  import java.util.Scanner;
2
3  /**
4     Esse programa calcula os valores médios e máximos de um conjunto
5     de valores de entrada.
6  */
7  public class DataAnalyzer
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12       DataSet data = new DataSet();
13
14       boolean done = false;
15       while (!done)
16       {
17          System.out.print("Enter value, Q to quit: ");
18          String input = in.next();
19          if (input.equalsIgnoreCase("Q"))
20             done = true;
21          else
22          {
23             double x = Double.parseDouble(input);
24             data.add(x);
25          }
26       }
27
28       System.out.println("Average = " + data.getAverage());
29       System.out.println("Maximum = " + data.getMaximum());
30    }
31 }
```

ch06/dataset/DataSet.java

```
1  /**
2     Calcula informações sobre um conjunto de valores de dados.
3  */
4  public class DataSet
5  {
6     /**
7        Constrói um conjunto de dados vazio.
8     */
9     public DataSet()
10    {
11       sum = 0;
12       count = 0;
13       maximum = 0;
14 }
15
```

```
   /**
      Adiciona um valor de dados ao conjunto de dados.
      @param x valor de dados
   */
   public void add(double x)
   {
      sum = sum + x;
      if (count == 0 || maximum < x) maximum = x;
      count++;
   }

   /**
      Obtém a média dos dados adicionados.
      @return média ou 0 se nenhum dado foi adicionado
   */
   public double getAverage()
   {
      if (count == 0) return 0;
      else return sum / count;
   }

   /**
      Obtém o maior dos dados adicionados.
      @return máximo ou 0 se nenhum dado foi adicionado
   */
   public double getMaximum()
   {
      return maximum;
   }

   private double sum;
   private double maximum;
   private int count;
}
```

Saída

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

7. Por que a classe `DataAnalyzer` chama `in.next` e não `in.nextDouble`?
8. Essa classe `DataSet` continuaria a calcular o valor máximo correto se você simplificasse a atualização do campo `maximum` no método `add` para esta instrução?

```
if (maximum < x) maximum = x;
```

COMO FAZER 6.1

Implementando laços

Você escreve um laço porque seu programa precisa repetir uma ação várias vezes. Como vimos neste capítulo, há diversos tipos de laço e nem sempre é óbvio como estruturar comandos deste tipo. Este Como Fazer o orienta no processo mental envolvido na programação de um laço.

Passo 1 Liste o trabalho que precisa ser feito em cada passo do corpo do laço.

Por exemplo, suponha que você precise ler os valores de entrada em galões e convertê-los em litros até que o final da entrada seja alcançado. As operações são:

- Ler a entrada
- Converter a entrada em litros
- Imprimir a resposta

Suponha que você precise verificar os caracteres de uma string e contar as vogais. As operações são:

- Obter o próximo caractere
- Se for uma vogal, incrementar um contador

Passo 2 Descobrir a freqüência com que o laço é repetido.

Respostas típicas poderiam ser:

- Dez vezes
- Uma vez para cada caractere na string
- Até o final da entrada ser alcançado
- Enquanto o saldo for menor que o saldo pretendido

Se um laço for executado um número definido de vezes, normalmente um laço `for` será adequado. As primeiras duas respostas acima levam a laços `for`, como

```
for (int i = 1; i <= 10; i++) . . .
for (int i = 0; i < str.length(); i++) . . .
```

As duas próximas precisam ser implementadas como laços `while` – você não sabe quantas vezes o corpo do laço será repetido.

Passo 3 Com um laço `while`, descubra onde você pode determinar se o laço terminou.

Há três possibilidades:

- Antes de entrar no laço
- No meio do laço
- No fim do laço

Por exemplo, se executar um laço enquanto o saldo é menor que o saldo alvo, você pode verificar essa condição no começo do laço. Se o saldo for menor que o saldo alvo, você entra no laço. Se não for, você terminou. Nesse caso, seu laço tem a forma:

```
while (condição)
{
   Faz o trabalho
}
```

Mas verificar a entrada requer que você primeiro *leia* essa entrada. Isso significa que precisará entrar no laço, ler a entrada e então decidir se quer ir mais longe. Seu laço tem a forma:

```
boolean done = false;
while (!done)
{
   Faz o trabalho necessário para verificar a condição
   if (condição)
      done = true;
   else
   {
      Faz outro trabalho
   }
}
```

Essa estrutura de laço às vezes é chamada *"laço e meio"*.

Por fim, se souber que precisa prosseguir depois de passar pelo laço uma vez, então utilize um laço `do/while`:

```
do
{
   Faz o trabalho
}
while (condição)
```

Mas esses laços são muito raros na prática.

Passo 4 Implementar o laço adicionando as operações do Passo 1 ao corpo do laço.

Ao escrever um laço `for`, normalmente utilizamos o índice de laço dentro do seu corpo. Por exemplo, "obtenha o próximo caractere" é implementado como a instrução:

```
char ch = str.charAt(i);
```

Passo 5 Verifique duas vezes as inicializações das suas variáveis.

Se utilizar uma variável booleana `done`, certifique-se de que ela é inicializada como `false`. Se acumular um resultado em uma variável `sum` ou `count`, certifique-se de configurá-la como 0 antes de entrar no laço pela primeira vez.

Passo 6 Verifique a possibilidade de erros "por um".

Pense nos cenários mais simples possíveis:

- Se você lê a entrada, o que acontece se não houver nenhuma entrada? E se houver exatamente uma entrada?
- Se examinar os caracteres de uma string, o que acontece se a string estiver vazia? E se ela tiver um caractere?
- Se acumular valores até que algum alvo tenha sido alcançado, o que acontece se o alvo for 0? E se for um valor negativo?

Investigue manualmente cada instrução no laço, incluindo todas as inicializações. Verifique cuidadosamente todas as condições, prestando atenção à diferença entre comparações como < e <=. Verifique que o laço não foi percorrido nem uma única vez e que o resultado final é aquele que você espera.

Se você escrever um laço `for`, verifique se seus limites devem ser simétricos ou assimétricos (veja a Dica de qualidade 6.3) e conte o número de iterações (veja a Dica de qualidade 6.4).

DICA DE QUALIDADE 6.3

Limites simétricos e assimétricos

É fácil escrever um laço com `i` indo de 1 a `n`:

```
for (i = 1; i <= n; i++) . . .
```

Os valores para `i` estão limitados pela relação $1 \leq i \leq n$. Como há comparações ≤ nos dois limites, esses limites são chamados *simétricos*.

Ao percorrer os caracteres em uma string, os limites são *assimétricos*.

```
for (i = 0; i < str.length(); i++) . . .
```

> Faça uma escolha entre laços de limites simétricos e assimétricos.

Os valores para `i` estão limitados por $0 \leq i < str.length()$, com uma comparação ≤ à esquerda e uma comparação < à direita. Isso é adequado porque `str.length()` não é uma posição válida.

Não é uma boa idéia forçar a simetria artificialmente:

```
for (i = 0; i <= str.length() - 1; i++) . . .
```

Isso é mais difícil de ler e de entender.

Para cada laço, considere qual forma é mais natural para o problema e a utilize.

DICA DE QUALIDADE 6.4

Conte as iterações

Localizar os limites superiores e inferiores corretos para uma iteração pode ser confuso. Devo iniciar em 0? Devo utilizar `<= b` ou `< b` como uma condição de término?

> Conte o número de iterações para verificar se seu laço `for` está correto.

Conte o número de iterações para verificar se seu laço `for` está correto. Contar o número de iterações é um recurso muito útil para melhor entender um laço. A contagem é mais fácil para laços com limites assimétricos. O laço

```
for (i = a; i < b; i++) . . .
```

é executado `b - a` vezes. Por exemplo, o laço que percorre os caracteres em uma string,

```
for (i = 0; i < str.length(); i++) . . .
```

executa `str.length()` vezes. Isso faz total sentido, porque há `str.length()` caracteres em uma string.

O laço com limites simétricos,

```
for (i = a; i <= b; i++)
```

é executado `b - a + 1` vezes. Esse "`+ 1`" é a fonte de muitos erros de programação. Por exemplo,

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

executa 11 vezes. Talvez seja isso o que você quer; do contrário, inicie em `1` ou utilize `< 10`.

Uma maneira de visualizar esse erro "`+ 1`" é pensar nas estacas e seções de uma cerca. Suponha que a cerca tenha dez seções (=). Quantas estacas (|) ela tem?

```
|=|=|=|=|=|=|=|=|=|=|
```

Uma cerca com dez seções tem *onze* estacas. Cada seção tem uma estaca à esquerda *e* há mais uma estaca depois da última seção. Esquecer de contar a última iteração de um laço "<=" é freqüentemente chamado "erro de estaca de cerca".

Se o incremento for um valor `c` diferente de 1 e `c` divide `b - a`, então as contagens serão

`(b - a) / c`	para o laço assimétrico
`(b - a) / c + 1`	para o laço simétrico

Por exemplo, o laço `for (i = 10; i <= 40; i += 5)` executa `(40 - 10)/5 + 1 = 7` vezes.

TÓPICO AVANÇADO 6.3

O problema do "um laço e meio"

O Tópico Avançado 6.3 discute duas estratégias alternativas para implementar um laço cuja condição de conclusão é determinada no meio do caminho dentro do corpo do laço.

TÓPICO AVANÇADO 6.4

Os comandos `break` e `continue`

O Tópico Avançado 6.4 discute os comandos `break` e `continue` opcionais. Nenhum destes comandos é necessário para implementar laços, mas ocasionalmente podem tornar um laço complexo mais conciso.

6.5 Números aleatórios e simulações

Em uma simulação, você gera repetidamente números aleatórios e usa-os para simular uma atividade.

Em uma simulação você gera eventos aleatórios e avalia os resultados. Eis um problema típico que pode ser decidido executando uma simulação: a *experiência da agulha de Buffon*, criada por Comte Georges-Louis Leclerc de Buffon (1707–1788), um naturalista francês. Em cada *tentativa*, uma agulha com comprimento de uma polegada (2,54 cm) é lançada sobre um papel pautado com linhas paralelas separadas por 2 polegadas. Se a agulha cair sobre uma linha, ela será contada como um sucesso. (Veja Figura 3.) Buffon presumiu que o quociente entre *tentativas/sucessos* se aproxima de π.

Figura 3
A experiência da agulha de Buffon.

Agora, como você pode executar essa experiência no computador? Você não quer realmente construir um robô que lance agulhas sobre um papel. A classe `Random` da biblioteca Java implementa um *gerador de números aleatórios* que produz números que parecem ser completamente aleatórios. Para gerar números aleatórios, você constrói um objeto da classe `Random` e então aplica um dos métodos a seguir:

Método	Retorna
`nextInt(n)`	Um inteiro aleatório entre os inteiros 0 (inclusive) e `n` (exclusive)
`nextDouble()`	Um número aleatório de ponto flutuante entre 0 (inclusive) e 1 (exclusive)

Por exemplo, você pode simular o lançamento de um dado desta maneira:

```
Random generator = new Random();
int d = 1 + generator.nextInt(6);
```

A chamada `generator.nextInt(6)` fornece um número aleatório entre 0 e 5 (inclusive). Adicione 1 para obter um número entre 1 e 6.

Se chamar `nextInt` dez vezes, você obterá uma seqüência de números aleatórios semelhante a esta:

```
6 5 6 3 2 6 3 4 4 1
```

Na verdade, os números não são completamente aleatórios. Eles são selecionados a partir de seqüências muito longas de números que não se repetem por muito tempo. Essas seqüências são calculadas a partir de fórmulas relativamente simples; elas se comportam como números aleatórios. Por essa razão, são freqüentemente chamadas de números pseudo-aleatórios. Gerar boas seqüências de números que se comportam como seqüências verdadeiramente aleatórias é um problema importante e muito estudado em ciência da computação. Não investigaremos essa questão em detalhes; simplesmente utilizaremos os números aleatórios produzidos pela classe `Random`.

Para executar a experiência da agulha de Buffon, precisamos trabalhar um pouco mais. Quando você lança um dado, um dos seis lados permanece virado para cima. Mas ao lançar uma agulha, porém, há vários resultados possíveis. Você deve gerar *dois* números aleatórios: um para descrever a posição inicial e outro para descrever o ângulo da agulha em relação ao eixo x. Você então precisa testar se a agulha toca uma linha da grade. Pare depois de 10.000 tentativas.

Vamos gerar o ponto *inferior* da agulha. A coordenada x é irrelevante e você pode supor que y_{low} da coordenada y seja qualquer número aleatório entre 0 e 2. Mas, uma vez que ela pode ser um número aleatório *de ponto flutuante*, utilizamos o método `nextDouble` da classe `Random`. Ele retorna um número aleatório de ponto flutuante entre 0 e 1. Multiplique por 2 para obter um número aleatório entre 0 e 2.

O ângulo α entre a agulha e o eixo x pode ser um valor qualquer entre 0 e 180 graus. A extremidade superior da agulha tem a seguinte coordenada y:

$$y_{high} = y_{low} + \sin(\alpha)$$

A agulha cai sobre uma das linhas se y_{high} for pelo menos 2. Veja Figura 4.

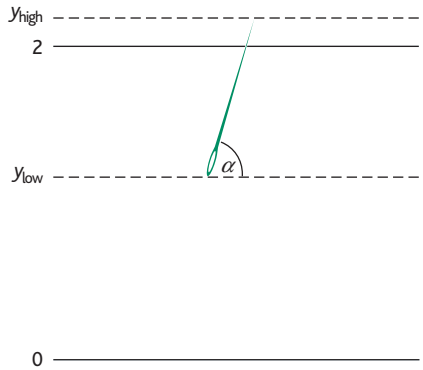


Figura 4
Quando a agulha cai sobre uma linha?

Para ver o programa que executa a simulação da experiência da agulha, consulte `ch06/random2/` no código-fonte disponível no material online do livro no site www.bookman.com.br.

O objetivo desse programa não é calcular π – há várias outras maneiras eficientes de fazer isso. Antes de tudo, o objetivo é demonstrar como uma experiência física pode ser simulada no computador. Buffon precisou lançar fisicamente a agulha milhares de vezes e registrar os resultados, o que deve ter sido bastante maçante. O computador pode executar essa experiência rápida e precisamente.

Simulações são aplicações de computador bem comuns. Muitas simulações utilizam essencialmente o mesmo padrão que o do código deste exemplo: em um laço, um grande número de valores de exemplo é gerado e os valores de certas observações são registrados para cada exemplo. Quando a simulação termina, são impressas as médias, ou outras estatísticas importantes, a partir dos valores observados.

Um exemplo típico de uma simulação é a modelagem de filas de clientes em um banco ou em um supermercado. Em vez de observar clientes reais, são simuladas as chegadas e as transações dos clientes no computador do caixa ou do balcão de pagamento. Pode-se testar no computador diferentes padrões de leiaute de edifícios ou de alocação de pessoal simplesmente fazendo alterações no programa. No mundo real, fazer essas alterações e medir seus efeitos seria impossível ou, pelo menos, muito caro.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

9. Como você usa um gerador de números aleatórios para simular o lançamento de uma moeda?
10. Por que o programa `NeedleSimulator` não é um método eficiente para calcular o π?

TÓPICO AVANÇADO 6.5

Invariantes de laço

O Tópico Avançado 6.5 mostra como você pode utilizar a técnica de invariantes de laço para provar que um laço sempre calculará o resultado correto.

FATO ALEATÓRIO 6.2

Provas de correção

O Fato Aleatório 6.2 discute a promessa e as limitações das provas de correção.

6.6 Utilizando um depurador

Como você certamente já deve ter percebido, programas de computador raramente executam perfeitamente na primeira vez. Às vezes, pode ser bem frustrante localizar os erros. Naturalmente, você pode inserir comandos para imprimir, executar o programa e tentar analisar a listagem impressa. Se essa listagem não apontar o problema claramente, talvez você precise adicionar e remover comandos para imprimir e executar o programa novamente. Isso pode ser um processo demorado.

Um depurador é um programa que você pode utilizar para executar outro programa e analisar o comportamento em tempo de execução.

Ambientes de desenvolvimento modernos contêm programas especiais, chamados depuradores, que ajudam a localizar erros permitindo que você acompanhe a execução de um programa. Você pode parar e reiniciar o programa e ver o conteúdo das variáveis sempre que seu programa for temporariamente interrompido. Em cada interrupção, você tem a escolha de quais variáveis inspecionar e quantos passos do programa executar até a próxima parada.

Algumas pessoas acham que depuradores são uma ferramenta que torna os programadores preguiçosos. De fato, algumas pessoas gostam de escrever programas grosseiros para então corrigi-los com um depurador, mas a maioria dos programadores faz um esforço honesto de escrever o melhor programa possível antes de tentar executá-lo em um depurador. Esses programadores percebem que um depurador, embora mais conveniente do que comandos de impressão, tem um preço. Leva tempo configurar e executar uma sessão eficaz de depuração.

Na prática, não há como evitar o uso de um depurador. Quanto maior o tamanho dos programas, mais difícil é depurá-los simplesmente inserindo comandos de impressão. Você descobrirá que o tempo necessário para entender um depurador é amplamente recompensado na sua carreira de programador.

Como ocorre com compiladores, depuradores variam bastante de um sistema para outro. Em alguns sistemas eles são bem primitivos e requerem que você memorize um pequeno conjunto de comandos obscuros; em outros eles têm uma interface intuitiva de janelas. As capturas de tela neste capítulo mostram o depurador no ambiente de desenvolvimento do Eclipse. Você pode fazer o download gratuito a partir do site da Eclipse Foundation [1]. Outros ambientes integrados, como o BlueJ, também incluem depuradores. Há um depurador independente e gratuito chamado JSwat disponível na página JSwat Graphical Java Debugger [2].

Você terá de descobrir como preparar um programa para depuração e como iniciar um depurador no seu sistema. Se utilizar um ambiente de desenvolvimento integrado que contém um editor, um compilador e um depurador, normalmente este passo será muito fácil. Simplesmente construa o programa da maneira normal e selecione um comando de

menu para iniciar a depuração. Em alguns sistemas, você precisará construir manualmente uma versão de depuração do seu programa e então invocar o depurador.

Você pode usar um depurador de modo eficiente dominando apenas três conceitos: pontos de interrupção, inspeção passo a passo e inspeção de variáveis.

Depois de iniciar o depurador, você pode ir bem longe usando apenas três comandos de depuração: "set breakpoint" (configurar pontos de interrupção), "single step" (inspeção passo a passo) e "inspect variable" (inspeção de variáveis). Os nomes, conjunto de teclas a precionar ou cliques com o mouse para esses comandos diferem bastante entre depuradores, mas todos suportam esses comandos básicos. Você pode descobri-los a partir da documentação, do manual no laboratório ou perguntando a uma pessoa que já utilizou o depurador.

Quando um depurador executa um programa, a execução é suspensa sempre que um ponto de interrupção é alcançado.

Quando você inicia o depurador, ele é executado à velocidade total até alcançar um *ponto de interrupção*. A execução então pára e o ponto de interrupção que causou a parada é exibido (veja Figura 5). Agora você pode inspecionar as variáveis e investigar o programa uma linha por vez ou continuar executando o programa em velocidade total até que ele alcance o próximo ponto de interrupção. Quando o programa termina, o depurador também pára.

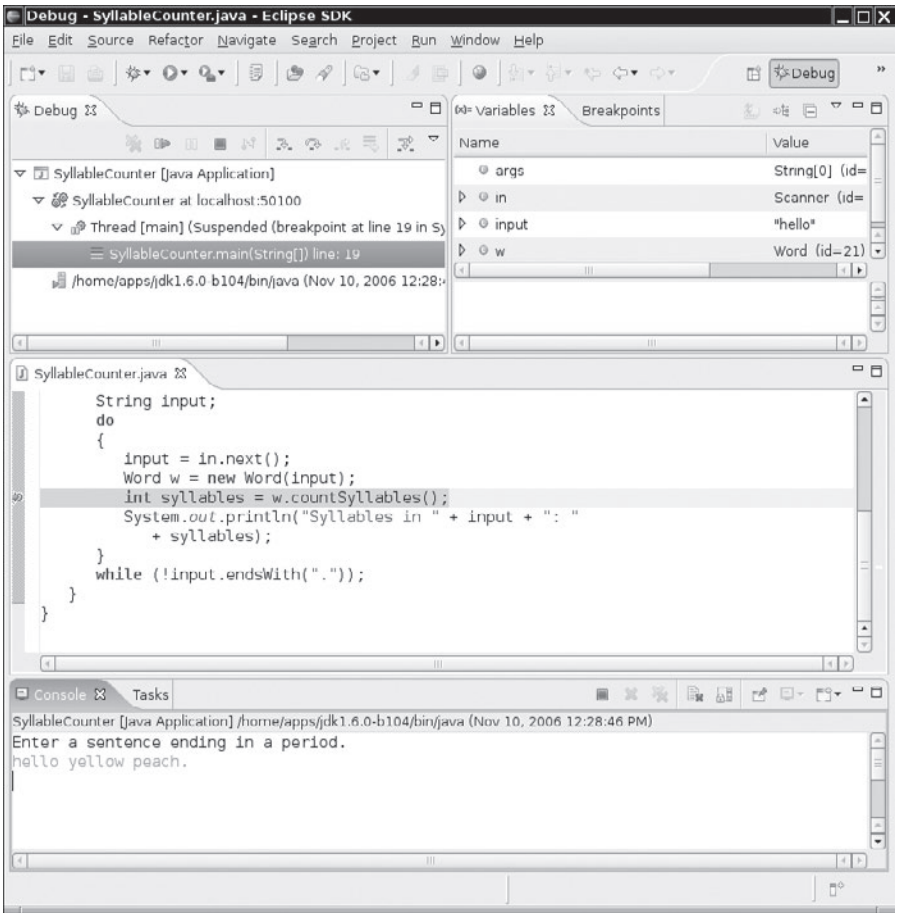


Figura 5 Parando em um ponto de interrupção.

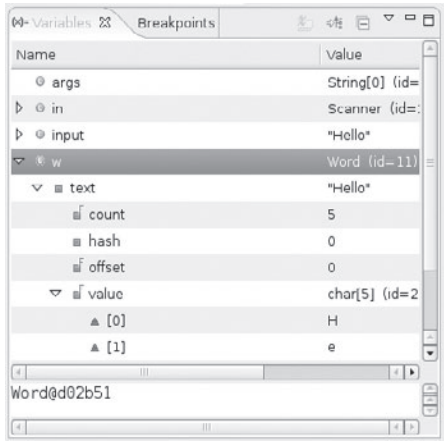


Figura 6
Inspecionando variáveis.

Pontos de interrupção permanecem ativos até você removê-los, portanto, você deve desativar periodicamente os pontos de interrupção que você não precisa mais.

Depois de o programa parar, você pode examinar os valores atuais das variáveis. Mais uma vez, o método para selecionar as variáveis difere entre depuradores. Alguns depuradores sempre exibem uma janela com as variáveis locais atuais. Em outros, você dá um comando como "inspecionar variável" e digita ou clica nela. O depurador então exibe o conteúdo dessa variável. Se todas as variáveis contiverem o esperado, você poderá executar o programa até o próximo ponto em que você quer parar.

Ao inspecionar objetos, freqüentemente você precisa dar um comando para "abrir" o objeto, por exemplo, clicando em um nó de uma árvore. Depois que o objeto é aberto, você vê suas variáveis de instância (veja Figura 6).

O comando single-step (inspeção passo a passo) executa o programa linha por linha.

Executar até um ponto de interrupção mostra rapidamente a linha que você quer, mas você não sabe como o programa chegou ali. Você também pode investigar o programa uma linha por vez. Assim, você entende o fluxo do programa, mas pode demorar bastante tempo para investigá-lo. O *comando single-step* executa a linha atual e pára na próxima linha do programa. A maioria dos depuradores tem dois comandos do tipo passo a passo, um chamado *step into* (entrar), que verifica as chamadas a métodos, e outro chamado *step over* (pular), que pula as chamadas a métodos.

Por exemplo, suponha que a linha atual seja:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

Quando você usa um comando *step over* para pular chamadas de método, você passa para a próxima linha:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

Mas se utilizar um comando *step into* para verificar as chamadas de método, você entra na primeira linha do método `countSyllables`.

```
public int countSyllables()
{
   int count = 0;
   int end = text.length() - 1;
   . . .
}
```

Você deve usar o comando *step into* em um método para verificar se ele executa seu trabalho corretamente. Você deve utilizar um comando *step over* em um método quando souber que ele funciona corretamente.

Por fim, quando a execução do programa pára, a sessão de depuração também pára. Para executar o programa novamente, talvez seja necessário reinicializar o depurador ou fechar o programa de depuração e iniciar novamente. Os detalhes dependem do depurador.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

11. No depurador, você alcança uma chamada a `System.out.println`. Você deve entrar (*step into*) ou pular (*step over*) o método?
12. No depurador, você alcança o começo de um método com alguns laços dentro. Você quer descobrir o valor de retorno calculado no final do método. Você deve configurar um ponto de interrupção ou percorrer passo a passo o método?

6.7 Uma sessão de depuração de exemplo

Para termos um exemplo realista da execução de um depurador, analisaremos uma classe `Word` cujo principal objetivo é contar o número de sílabas em uma palavra. A classe usa esta regra para contar sílabas:

Cada grupo de vogais adjacentes na língua inglesa (a, e, i, o, u, y) é contada como uma sílaba (por exemplo, "ea" em "peach" é uma sílaba, mas o "e . . . o" em "yellow" é contado como duas sílabas). Mas um "e" no final de uma palavra não é contado como uma sílaba. Cada palavra tem pelo menos uma sílaba, ainda que as regras anteriores forneçam uma contagem de 0.

Além disso, ao construir uma palavra a partir de uma string, todos os caracteres no início ou no final da string que não são letras são removidos. Isso é útil quando você lê a entrada utilizando o método `next` da classe `Scanner`. Strings de entrada ainda podem conter aspas e marcas de pontuação, e não queremos que elas façam parte da palavra.

Eis o código-fonte. Há alguns erros nesta classe.

ch06/debugger/Word.java

```
1  public class Word
2  {
3     /**
4        Constrói uma palavra removendo
5        caracteres iniciais e finais que não são letras, como sinais de pontuação.
6        @param s string de entrada
7     */
8     public Word(String s)
```

```
09    {
10       int i = 0;
11       while (i < s.length() && !Character.isLetter(s.charAt(i)))
12          i++;
13       int j = s.length() - 1;
14       while (j > i && !Character.isLetter(s.charAt(j)))
15          j--;
16       text = s.substring(i, j);
17    }
18
19    /**
20       Retorna o texto da palavra, depois da remoção dos
21       caracteres iniciais e finais que não são letras.
22       @return texto da palavra
23    */
24    public String getText()
25    {
26       return text;
27    }
28
29    /**
30       Conta as sílabas na palavra.
31       @return contagem de sílabas
32    */
33    public int countSyllables()
34    {
35       int count = 0;
36       int end = text.length() - 1;
37       if (end < 0) return 0; // A string vazia não contém nenhuma sílaba
38
39       // Um e no final da palavra não conta como uma vogal
40       char ch = Character.toLowerCase(text.charAt(end));
41       if (ch == 'e') end--;
42
43       boolean insideVowelGroup = false;
44       for (int i = 0; i <= end; i++)
45       {
46          ch = Character.toLowerCase(text.charAt(i));
47          if ("aeiouy".indexOf(ch) >= 0)
48          {
49             // ch é uma vogal
50             if (!insideVowelGroup)
51             {
52                // Início do novo grupo de vogais
53                count++;
54                insideVowelGroup = true;
55             }
56          }
57       }
58
59       // Cada palavra tem pelo menos uma sílaba
60       if (count == 0)
61          count = 1;
62
```

```
63         return count;
64      }
65
66      private String text;
67   }
```

Eis uma classe simples de teste. Digite uma frase e as contagens de sílabas de todas as palavras são exibidas.

ch06/debugger/SyllableCounter.java

```
1   import java.util.Scanner;
2
3   /**
4      Esse programa conta as sílabas de todas as palavras em uma frase.
5   */
6   public class SyllableCounter
7   {
8      public static void main(String[] args)
9      {
10        Scanner in = new Scanner(System.in);
11
12        System.out.println("Enter a sentence ending in a period.");
13
14        String input;
15        do
16        {
17           input = in.next();
18           Word w = new Word(input);
19           int syllables = w.countSyllables();
20           System.out.println("Syllables in " + input + ": "
21                 + syllables);
22        }
23        while (!input.endsWith("."));
24     }
25  }
```

Forneça esta entrada:

```
hello yellow peach.
```

Então a saída será:

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

Isso não é muito promissor.

Primeiro, configure um ponto de interrupção na primeira linha do método `countSyllables` da classe `Word`, na linha 33 de `Word.java`. Então, inicie o programa. Ele solicitará a entrada. O programa irá parar no ponto de interrupção que você acabou de configurar.

```
SyllableCounter.java   Word.java
    @return the syllable count
*/
public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    if (end < 0) return 0; // The empty string has no syllables

    // An e at the end of the word doesn't count as a vowel
    char ch = Character.toLowerCase(text.charAt(end));
    if (ch == 'e') end--;

    boolean insideVowelGroup = false;
```

Figura 7 Depurando o método `countSyllables`.

Primeiro, o método `countSyllables` verifica se o último caractere da palavra é uma letra `'e'`. Vamos verificar se isso funciona corretamente. Execute o programa até a linha 41 (veja Figura 7).

Agora, inspecione a variável `ch`. Esse depurador em particular tem uma exibição útil de todas as variáveis locais e de instância atuais – veja Figura 8. Se o seu depurador não tiver, talvez você precise inspecionar `ch` manualmente. Você pode ver que `ch` contém o valor `'l'`. Isso é estranho. Examine o código-fonte. A variável `end` foi configurada como `text.length() - 1`, a última posição na string `text`, e `ch` é o caractere nessa posição.

Examinando mais detalhadamente, você descobrirá que `end` foi configurada como 3, não 4, como você poderia esperar. E `text` contém a string `"hell"`, não `"hello"`. Portanto, não causa surpresa que `countSyllables` retorne a resposta 1. Precisaremos procurar o culpado em outra parte. Aparentemente, o construtor `Word` contém um erro.

Infelizmente, um depurador não pode voltar no tempo. Portanto, você precisa parar o depurador, configurar um ponto de interrupção no construtor `Word` e reiniciar o depurador. Forneça a entrada mais uma vez. O depurador irá parar no começo do construtor `Word`. O construtor configura duas variáveis `i` e `j`, pulando todos os caracteres que não são letras no início e no final da string de entrada. Configure um ponto de interrupção depois do fim do segundo laço (veja Figura 9) para inspecionar os valores de `i` e `j`.

Variables | Breakpoints

Name	Value
this	Word (id=11)
text	"hell"
count	0
end	3
ch	l

Figura 8 Valores atuais das variáveis locais e de instância.

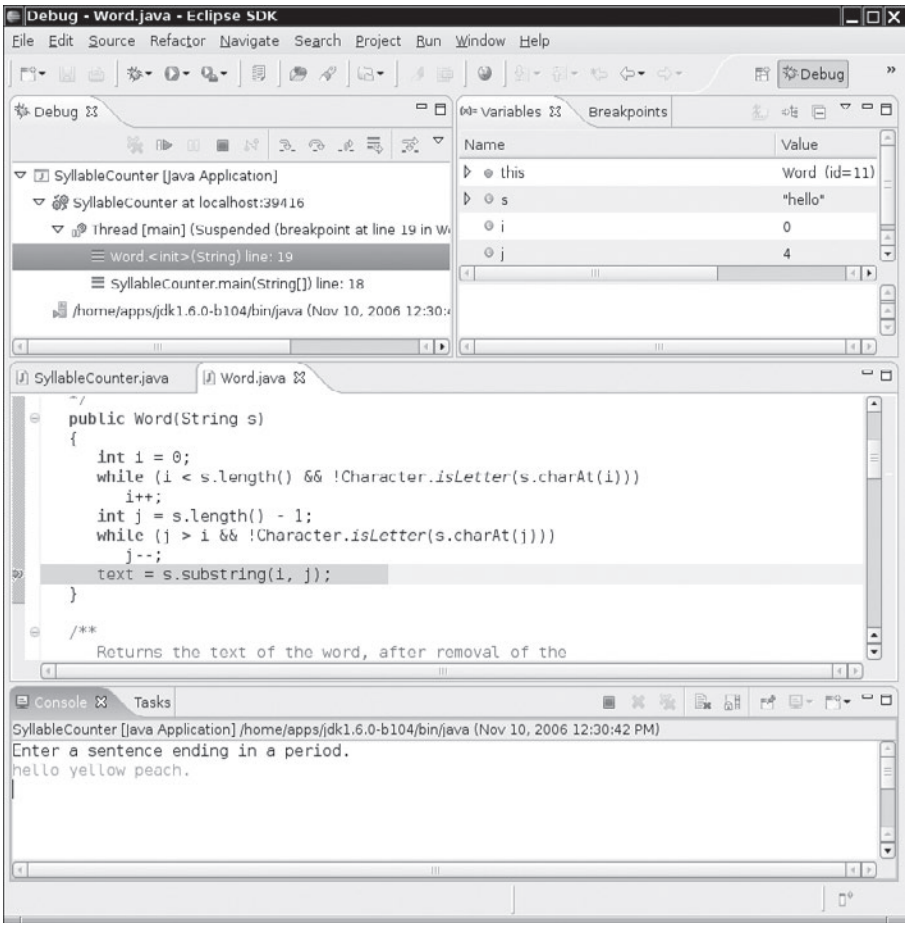


Figura 9 Depurando o construtor `Word`.

Nesse ponto, inspecionar `i` e `j` mostra que `i` é 0 e `j` é 4. Isso faz sentido – não há nenhum sinal de pontuação a pular. Portanto, por que `text` é configurado como `"hell"`? Lembre-se de que o método `substring` conta as posições até, mas sem incluir, o segundo parâmetro. Portanto, a chamada correta deve ser:

```
text = s.substring(i, j + 1);
```

Esse é um erro "de um" muito comum.

Corrija esse erro, recompile o programa e teste os três casos de teste novamente. Agora você obterá a seguinte saída:

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

Como você pode ver, ainda há um problema. Apague todos os pontos de interrupção e configure um novo ponto de interrupção no método `countSyllables`. Inicie o depurador e forneça a entrada `"hello."`. Quando o depurador parar no ponto de interrupção,

inicie uma verificação passo a passo das linhas do método. Eis o código do laço que conta as sílabas:

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
   ch = Character.toLowerCase(text.charAt(i));
   if ("aeiouy".indexOf(ch) >= 0)
   {
      // ch é uma vogal
      if (!insideVowelGroup)
      {
         // Início do novo grupo de vogais
         count++;
         insideVowelGroup = true;
      }
   }
}
```

Na primeira iteração pelo laço, o depurador pula o comando `if`. Isso faz sentido, porque a primeira letra, `'h'`, não é uma vogal. Na segunda iteração, o depurador entra no comando `if`, como deveria, porque a segunda letra, `'e'`, é uma vogal. A variável `insideVowelGroup` é configurada como `true` e o contador de vogais é incrementado. Na terceira iteração, o comando `if` é novamente pulado, porque a letra `'l'` não é uma vogal. Mas na quinta iteração, algo estranho acontece. A letra `'o'` é uma vogal e o fluxo do programa entra no comando `if`. Mas o segundo comando `if` é pulado e `count` não é incrementado novamente.

Por quê? A variável `insideVowelGroup` ainda é verdadeira, ainda que o primeiro grupo de vogais tenha terminado quando a consoante `'l'` foi encontrada. Ler uma consoante deve configurar `insideVowelGroup` de novo como `false`. Esse é um erro de lógica mais sutil, mas não é incomum ao projetar um laço que monitora o estado do processamento. Para corrigi-lo, pare o depurador e adicione a cláusula a seguir:

```
if ("aeiouy".indexOf(ch) >= 0)
{
   . . .
}
else insideVowelGroup = false;
```

Um depurador só pode ser utilizado para analisar a presença de erros, não para mostrar que um programa está livre deles.

Agora recompile e execute o teste mais uma vez. A saída é:

```
Syllables in hello: 2
Syllables in yellow: 2
Syllables in peach.: 1
```

O programa agora está livre de erros? Essa não é uma pergunta que o depurador pode responder. Lembre-se: o teste só pode mostrar a presença de erros, não sua ausência.

Autoverificação da aprendizagem

13. O que causou o primeiro erro encontrado nesta sessão de depuração?

14. O que causou o segundo erro? Como ele foi detectado?

COMO FAZER 6.2

Depurando

Agora você conhece a mecânica da depuração, mas todo esse conhecimento ainda pode deixá-lo desamparado ao executar um depurador para examinar um programa com problemas. Há algumas estratégias que você pode utilizar para reconhecer erros e suas causas.

Passo 1 Reproduza o erro.

À medida que você testa seu programa, você percebe que às vezes ele faz algo errado. Ele fornece a saída errada, parece imprimir algo completamente aleatório, entra em um laço infinito ou trava. Descubra exatamente como reproduzir esse comportamento. Que números você inseriu? Onde você clicou?

Execute o programa novamente; digite exatamente as mesmas respostas e clique nas mesmas áreas (ou o mais perto que você puder). O programa exibe o mesmo comportamento? Se sim, então faz sentido executar um depurador para analisar esse problema específico. Depuradores são bons para analisar falhas específicas. Eles não são muito úteis para analisar o contexto geral de um programa.

Passo 2 Simplifique o erro.

Antes de você executar um depurador, é interessante investir alguns minutos para tentar descobrir uma entrada mais simples que também produza um erro. Você pode utilizar palavras mais curtas ou números mais simples e mesmo assim o programa não funciona corretamente? Se sim, utilize esses valores durante sua sessão de depuração.

Passo 3 Divida para conquistar.

> Utilize a técnica de dividir para conquistar a fim de localizar o ponto de falha de um programa.

Agora que há uma falha particular, você quer chegar o mais perto possível dela. O ponto-chave da depuração é localizar o código que produz a falha. Assim como com as pragas do mundo real, localizar um erro pode ser difícil, mas depois de encontrá-lo, eliminá-lo normalmente é a parte fácil. Suponha que uma divisão por 0 faça seu programa ser fechado. Como há muitas operações de divisão em um programa típico, muitas vezes não é viável configurar pontos de interrupção para todas elas. Em vez disso, utilize uma técnica de dividir para conquistar. Pule os métodos em `main` sem verificar falhas. Com o tempo, a falha acontecerá novamente. Agora você sabe qual método contém o erro: É o último método que foi chamado a partir de `main` antes de o programa ser fechado. Reinicie o depurador, volte a essa linha e então verifique o método `main`. Repita o processo.

Por fim, você terá localizado a linha que contém a divisão inválida. Talvez seja completamente óbvio, a partir do código, descobrir por que o denominador não está correto. Se não estiver, você precisará encontrar o local em que ele é calculado. Infelizmente, você não pode voltar atrás no depurador. Você precisa reiniciar o programa e ir para o ponto em que o cálculo do denominador acontece.

Passo 4 Conheça o que seu programa deve fazer.

> Durante a depuração, compare o conteúdo real das variáveis com os valores que você sabe que deveriam ter.

O depurador mostra o que o programa faz. Você precisa entender o que o programa *deve* fazer ou não será capaz de localizar erros. Antes de investigar um laço, pense em quantas iterações você

espera que o programa crie. Antes de inspecionar uma variável, pense no que você espera ver. Se não tiver pistas, pare um pouco e pense. Tenha uma calculadora à mão para que possa fazer seus próprios cálculos. Se você souber qual é o valor, inspecione a variável. Esse é o momento da verdade. Se o programa continuar na trilha certa, esse será o valor esperado e você precisa procurar o erro com mais detalhes. Se o valor for diferente, você pode estar no caminho de algo. Verifique duas vezes seu cálculo. Se tiver certeza de que o valor está correto, descubra por que seu programa propõe um valor diferente.

Em muitos casos, os erros de programa são o resultado de erros simples como condições de término de laço que estão "por um". Mas o que mais acontece é os programas cometerem erros computacionais. Talvez eles devessem adicionar dois números, mas, acidentalmente, o código foi escrito para subtraí-los. Diferentemente do seu professor de cálculo, programas não fazem um esforço especial para assegurar que tudo seja um inteiro simples (e isso também não ocorre com problemas do mundo real). Você precisará fazer alguns cálculos com inteiros grandes ou números de ponto flutuante detestáveis. Às vezes, esses cálculos podem ser evitados se você simplesmente pensar, "Essa quantia deve ser positiva? Ela deve ser maior do que esse valor?" Em seguida, inspecione as variáveis para verificar suas teorias.

Passo 5 Examine todos os detalhes.

Ao depurar um programa, freqüentemente você tem uma teoria sobre a natureza do problema. Contudo, tenha uma mente aberta e examine todos os detalhes. Quais mensagens estranhas são exibidas? Por que o programa executa uma outra ação inesperada? Esses detalhes são importantes. Ao executar uma sessão de depuração, você é um detetive que precisa examinar todas as pistas disponíveis.

Se perceber uma outra falha no problema que você está em vias de solucionar, não diga simplesmente, "Faço isso mais tarde". Essa falha pode muito bem ser a causa original do problema atual. É melhor fazer uma anotação desse problema, corrigir o que você acabou de encontrar e então retornar à missão original.

Passo 6 Certifique-se de que você entende cada erro antes de corrigi-lo.

Depois de descobrir que um laço faz iterações excessivas, é muito tentador aplicar uma solução do tipo "band-aid" e subtrair 1 de uma variável para que o problema específico não reapareça. Há uma grande probabilidade de que essa correção rápida crie problemas em outras partes. Você precisa ter um entendimento completo de como o programa deve ser escrito antes de aplicar uma correção.

Às vezes você localiza uma série de erros, aplica uma série de correções e o problema simplesmente permanece. Isso normalmente é um sintoma de um problema maior relacionado à lógica do programa. Há pouca coisa que você pode fazer com o depurador. Você precisa repensar o projeto do programa e reorganizá-lo.

Fato aleatório 6.3

O primeiro bug

O Fato Aleatório 6.3 comenta a história do primeiro bug de computador; uma mariposa ficou presa em um relé no Mark II, um enorme computador eletromecânico na Universidade de Harvard, em 1947.

RESUMO DO CAPÍTULO

1. Um comando `while` executa um bloco de código repetidamente. Uma condição controla a freqüência com que o laço é executado.
2. Um erro "por um" é um erro comum ao programar laços. Pense através de casos de teste simples para evitar esse tipo de erro.
3. Utilize um laço `for` quando uma variável é executada a partir de um valor inicial até um valor final com um incremento ou decremento constante.
4. Laços podem ser aninhados. Um exemplo típico de laços aninhados é imprimir uma tabela com linhas e colunas.
5. Às vezes, a condição de conclusão de um laço só pode ser avaliada no meio de um laço. Você pode introduzir uma variável booleana para controlar esse tipo de laço.
6. Faça uma escolha entre laços de limites simétricos e assimétricos.
7. Conte o número de iterações para verificar se seu laço `for` está correto.
8. Em uma simulação, você gera números aleatórios repetidamente e usa-os para simular uma atividade.
9. Um depurador é um programa que você pode utilizar para executar outro programa e analisar o seu comportamento em tempo de execução.
10. Você pode usar um depurador eficientemente dominando apenas três conceitos: pontos de interrupção, inspeção passo a passo e inspeção de variáveis.
11. Quando um depurador executa um programa, a execução é suspensa sempre que um ponto de interrupção é alcançado.
12. O comando single-step (inspeção passo a passo) executa o programa linha por linha.
13. Um depurador só pode ser utilizado para analisar a presença de erros, não para mostrar que um programa está livre de erros.
14. Utilize a técnica de dividir para conquistar a fim de localizar o ponto de falha de um programa.
15. Durante a depuração, compare o conteúdo real das variáveis com os valores que você sabe que elas deveriam ter.

LEITURA ADICIONAL

1. `http://eclipse.org` Site da Eclipse Foundation.
2. `http://www.bluemarsh.com/java/jswat` Página Web JSwat Graphical Java Debugger.
3. Kai Lai Chung, *Elementary Probability Theory with Stochastic Processes*, Undergraduate Texts in Mathematics, Springer-Verlag, 1974.
4. Rudolf Flesch, *How to Write Plain English*, Barnes & Noble Books, 1979.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.util.Random
   nextDouble
   nextInt
```

EXERCÍCIOS DE REVISÃO

★★ **Exercício R6.1.** Que comandos de laço o Java suporta? Forneça regras simples para quando utilizar cada tipo de laço.

★★ **Exercício R6.2.** O que o seguinte código imprime?

```
for (int i = 0; i < 10; i++)
{
   for (int j = 0; j < 10; j++)
      System.out.print(i * j % 10);
   System.out.println();
}
```

★★ **Exercício R6.3.** Com que freqüência os seguintes laços são executados? Suponha que `i` seja uma variável do tipo inteiro que não é alterada no corpo do laço.

a. `for (i = 1; i <= 10; i++) . . .`

b. `for (i = 0; i < 10; i++) . . .`

c. `for (i = 10; i > 0; i--) . . .`

d. `for (i = -10; i <= 10; i++) . . .`

e. `for (i = 10; i >= 0; i++) . . .`

f. `for (i = -10; i <= 10; i = i + 2) . . .`

g. `for (i = -10; i <= 10; i = i + 3) . . .`

★ **Exercício R6.4.** Transforme o laço `for` a seguir em um laço `while`.

```
int s = 0;
for (int i = 1; i <= 10; i++) s = s + i;
```

★★ **Exercício R6.5.** Transforme o laço `do` a seguir em um laço `while`.

```
int n = 1;
double x = 0;
double s;
do
{
   s = 1.0 / (n * n);
   x = x + s;
   n++;
}
while (s > 0.01);
```

★ **Exercício R6.6.** O que é um laço infinito? No seu computador, como você pode terminar um programa que executa um laço infinito?

★★★ **Exercício R6.7.** Forneça três estratégias para implementar este "laço e meio":

```
laço
{
   Lê o nome da ponte
   Se não OK, fecha o laço
```

```
    Lê o comprimento da ponte em pés
    Se não OK, fecha o laço
    Converte o comprimento em metros
    Imprime os dados da ponte
}
```

Utilize uma variável booleana, uma instrução `break` e um método com múltiplas instruções `return`. Qual dessas três abordagens você acha que é a mais clara?

★ **Exercício R6.8.** Implemente um laço que pede para um usuário inserir um número entre 1 e 10, dando três tentativas para obter o valor correto.

★ **Exercício R6.9.** Às vezes os estudantes escrevem programas com instruções como "Insira dados, 0 para encerrar" que saem do laço de entrada de dados quando o usuário insere o número 0. Explique por que normalmente isso não é uma boa idéia.

★ **Exercício R6.10.** Como você utilizaria um gerador de números aleatórios para simular a representação de uma carta de baralho?

★ **Exercício R6.11.** O que é um erro do tipo "por um"? Forneça um exemplo a partir da sua própria experiência de programação.

★★ **Exercício R6.12.** Forneça um exemplo de um laço `for` no qual limites simétricos são mais naturais. Forneça um exemplo de um laço `for` no qual os limites assimétricos são mais naturais.

★ **Exercício R6.13.** O que são laços aninhados? Forneça um exemplo de onde um laço aninhado geralmente é utilizado.

★T **Exercício R6.14.** Explique as diferenças entre estas operações de um depurador sobre um método:

- Step into
- Step over

★★T **Exercício R6.15.** Explique em detalhes como inspecionar a string armazenada em um objeto `String` no seu depurador.

★★T **Exercício R6.16.** Explique em detalhes como inspecionar as informações armazenadas em um objeto `Rectangle` no seu depurador.

★★T **Exercício R6.17.** Explique em detalhes como utilizar seu depurador para inspecionar o saldo armazenado em um objeto `BankAccount`.

★★T **Exercício R6.18.** Explique a estratégia de dividir para conquistar para se aproximar de um bug em um depurador.

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P6.1.** *Conversão de moedas.* Escreva um programa `CurrencyConverter` que pede para o usuário inserir a taxa de câmbio atual entre o dólar americano e o euro. O programa lê os valores em dólares americanos e os converte em euros. Pare quando o usuário inserir `Q`.

★★★ **Exercício P6.2.** *Vôo de um projétil.* Suponha que a bala de um canhão seja impulsionada verticalmente a uma velocidade inicial v_0. Qualquer livro de matemática informará que a posição da bala após t segundos é $s(t) = -0.5 \cdot g \cdot t^2 + v_0 \cdot t$, onde $g = 9.81$ m/s^2 é a força gravitacional da Terra. Nenhum livro de matemática menciona por que alguém iria querer conduzir essa experiência obviamente perigosa, portanto, queremos realizá-la na segurança do computador.

Na realidade, confirmaremos o teorema a partir do cálculo por meio de uma simulação. Na nossa simulação, consideraremos a maneira como a bala se move em intervalos muito curtos de tempo Δt. Em um curto intervalo de tempo, a velocidade v torna-se quase constante e, assim, podemos calcular a distância dos movimentos da bala como $\Delta s = v \cdot \Delta t$. No nosso programa, simplesmente configuraremos:

```
double deltaT = 0.01;
```

e atualizaremos a posição por:

```
s = s + v * deltaT;
```

A velocidade muda constantemente – de fato, ela é reduzida pela força gravitacional da terra. Em um curto intervalo de tempo, v diminui por $g \cdot \Delta t$ e devemos manter a velocidade atualizada como:

```
v = v - g * deltaT;
```

Na próxima iteração a nova velocidade é utilizada para atualizar a distância.

Agora execute a simulação até a bala de canhão perder a propulsão e começar a cair. Obtenha a velocidade inicial como uma entrada (100 m/s é um bom valor). Atualize a posição e a velocidade 100 vezes por segundo, mas só imprima a posição a cada segundo inteiro. Também imprima os valores da fórmula exata $s(t) = -0.5 \cdot g \cdot t^2 + v_0 \cdot t$ para comparação. Utilize uma classe `Cannonball`.

Qual é o benefício desse tipo de simulação quando há uma fórmula exata disponível? Bem, a fórmula do livro de matemática *não* é exata. Na verdade, a força gravitacional diminui de acordo com a distância da bala de canhão em relação à superfície da Terra. Isso complica bastante a álgebra até o ponto em que não é possível fornecer uma fórmula exata para o movimento real, mas a simulação em um computador pode ser simplesmente estendida para aplicar uma força gravitacional variável. Para balas de canhão, a fórmula do livro de matemática é realmente satisfatória, mas os computadores são necessários para calcular trajetórias exatas para objetos que voam tão alto como mísseis balísticos.

★★ **Exercício P6.3.** Escreva um programa que imprime as seguintes potências de dez

```
1.0
10.0
100.0
1000.0
10000.0
100000.0
1000000.0
1.0E7
1.0E8
1.0E9
1.0E10
1.0E11
```

Implemente a seguinte classe:

```
public class PowerGenerator
{
   /**
      Constrói um gerador de potências.
      @param aFactor número que será multiplicado por ele mesmo
   */
   public PowerGenerator(double aFactor) { . . . }
   /**
      Calcula a próxima potência.
   */
   public double nextPower() { . . . }
   . . .
}
```

Forneça então uma classe de teste `PowerGeneratorRunner` que chama `System.out.println(myGenerator.nextPower())` doze vezes.

★★ **Exercício P6.4.** A *seqüência de Fibonacci* é definida pela regra a seguir. Os dois primeiros valores na seqüência são 1 e 1. Cada valor subseqüente é a soma dos dois valores que o precedem. Por exemplo, o terceiro valor é 1 + 1 = 2, o quarto valor é 1 + 2 = 3 e o quinto é 2 + 3 = 5. Se f_n representa o *n-ésimo* valor na seqüência de Fibonacci, então:

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{se } n > 2$$

Escreva um programa que pede para o usuário informar o número n e imprime os primeiros n valores na seqüência de Fibonacci. Utilize uma classe `FibonacciGenerator` com um método `nextNumber`.

Dica: Não há necessidade de armazenar todos os valores para f_n. Você só precisa dos dois últimos valores para calcular o próximo na seqüência:

```
fold1 = 1;
fold2 = 1;
fnew = fold1 + fold2;
```

Depois desse valor, descarte `fold2`, que não é mais necessário, e configure `fold2` como `fold1` e `fold1` como `fnew`.

Sua classe geradora será testada com este programa:

```
public class FibonacciRunner
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.println("Enter n:");
      int n = in.nextInt();

      FibonacciGenerator fg = new FibonacciGenerator();

      for (int i = 1; i <= n; i++)
         System.out.println(fg.nextNumber());
   }
}
```

★★ **Exercício P6.5.** *Média e desvio padrão.* Escreva um programa que lê um conjunto de valores de ponto flutuante a partir da entrada. Quando o usuário indicar o fim da entrada, imprima a contagem dos valores, a média e o desvio padrão. A média de um conjunto de dados $x_1, \ldots, x_n$ é

$$\bar{x} = \frac{\sum x_i}{n}$$

onde $\sum x_i = x_1 + \cdots + x_n$ é a soma dos valores de entrada. O desvio padrão é

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

Entretanto, essa fórmula não é adequada para nossa tarefa. No momento em que você tiver calculado a média, os x_i individuais terão desaparecidos. Até você entender como salvar esses valores, utilize a fórmula numericamente menos estável

$$s = \sqrt{\frac{\sum x_i^2 - \frac{1}{n}\left(\sum x_i\right)^2}{n - 1}}$$

Você pode calcular essa quantidade mantendo a contagem, a soma e a soma dos quadrados na classe `DataSet` à medida que processa os valores de entrada.

★★ **Exercício P6.6.** *Fatoração de inteiros.* Escreva um programa que pede para o usuário inserir um inteiro e então imprime todos os seus fatores em ordem ascendente. Por exemplo, quando o usuário insere 150, o programa deve imprimir:

```
2
3
5
5
```

Utilize uma classe `FactorGenerator` com um construtor `FactorGenerator(int numberToFactor)` e métodos `nextFactor` e `hasMoreFactors`. Forneça uma classe `FactorPrinter` cujo método `main` lê uma entrada de usuário, constrói um objeto `FactorGenerator` e imprime os fatores.

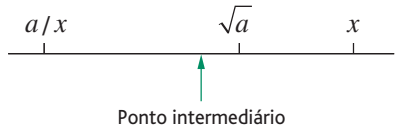
★★ **Exercício P6.7.** *Números primos.* Escreva um programa que pede para o usuário inserir um inteiro e então imprime todos os números primos até esse inteiro. Por exemplo, quando o usuário insere 20, o programa deve imprimir:

```
2
3
5
7
11
13
17
19
```

Lembre-se de que um número é primo se ele não for divisível por qualquer número, exceto por 1 e por ele mesmo.

Forneça uma classe `PrimeGenerator` com um método `nextPrime`.

★★ **Exercício P6.8.** O *método de Heron* é um método para calcular raízes quadradas que já era conhecido na Grécia antiga. Se x for uma suposição para o valor $\sqrt{a}$, então a média de x e a/x é uma suposição melhor.



Implemente uma classe `RootApproximator` que começa com uma suposição inicial de 1 e cujo método `nextGuess` produz uma seqüência de suposições cada vez melhor. Forneça um método `hasMoreGuesses` que retorna `false` se duas suposições sucessivas estiverem suficientemente próximas uma da outra (isto é, elas se diferem por não mais que um pequeno valor ε). Teste então sua classe desta maneira:

```
RootApproximator approx = new RootApproximator(a, EPSILON);
while (approx.hasMoreGuesses())
   System.out.println(approx.nextGuess());
```

★★ **Exercício P6.9.** O método iterativo mais conhecido para calcular as raízes de uma função f (isto é, os valores x para os quais $f(x)$ é 0) é a aproximação de Newton–Raphson. Para encontrar o zero de uma função cuja derivada também é conhecida, calcule:

$$x_{\text{new}} = x_{\text{old}} - f\left(x_{\text{old}}\right) / f'\left(x_{\text{old}}\right)$$

Para este exercício, escreva um programa para calcular as n-ésimas raízes de números de ponto flutuante. Solicite ao usuário a e n, então obtenha $\sqrt[n]{a}$ calculando um zero da função $f(x) = x^n - a$. Siga a abordagem do Exercício P6.8.

★★ **Exercício P6.10.** O valor de e^x pode ser calculado como a série de potências

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

onde $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

Escreva um programa que calcula e^x utilizando essa equação. É claro que você não pode calcular uma soma infinita. Simplesmente continue adicionando valores até que um número na soma individual (termo) seja menor que um certo limiar. Em cada passo, você precisa calcular o novo termo e adicioná-lo ao total. Atualize esses termos assim:

```
term = term * x / n;
```

Siga a abordagem dos dois exercícios anteriores, implementando uma classe `ExpApproximator`. Sua primeira suposição deve ser 1.

★ **Exercício P6.11.** Escreva um programa `RandomDataAnalyzer` que gere 100 números aleatórios entre 0 e 1000 e adicione-os a um `DataSet`. Imprima os valores médio e máximo.

★★ **Exercício P6.12.** Programe a simulação a seguir: dardos são lançados em pontos aleatórios sobre o quadrado com cantos (1,1) e (–1,–1). Se o dardo cair dentro do círculo de unidade (isto é, o círculo com centro (0,0) e raio 1), será computado um acerto. Caso contrário, será um erro. Execute essa simulação e utilize-a para determinar um valor aproximado para π. Você ganha um crédito extra se explicar por que esse é um método melhor para estimar π do que o programa da agulha de Buffon.

★★★G **Exercício P6.13.** *Passeio aleatório.* Simule o andar errante de uma pessoa embriagada utilizando uma grade para representar um quarteirão. Desenhe uma grade com 20 ruas horizontais e 20 ruas verticais. Represente a simulação do bêbado com um ponto posicionado inicialmente no meio da grade. Por 100 vezes, faça o bêbado simulado selecionar aleatoriamente uma direção (leste, oeste, norte, sul), mover-se um quarteirão na direção escolhida e desenhe o ponto. (Poderíamos esperar que, em média, talvez a pessoa não chegue a lugar algum porque, com o tempo, os movimentos para diferentes direções anulam uns aos outros, mas, na realidade, podemos demonstrar com a probabilidade de 1 que a pessoa acabará saindo de qualquer região finita.) Utilize classes para a grade e para o bêbado.

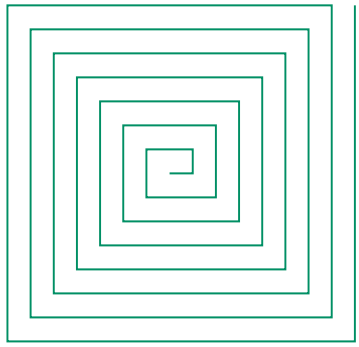
★★★G **Exercício P6.14.** Este exercício é uma continuação do Exercício P6.2. A maioria das balas de canhão não é disparada diretamente para cima, mas em um ângulo. Se a velocidade inicial tiver magnitude v e o ângulo inicial for α, então a velocidade será um vetor com componentes $v_x = v \cdot \cos(\alpha)$, $v_y = v \cdot \sin(\alpha)$. Na direção x a velocidade não muda. Na direção y a força gravitacional mostra seu efeito. Repita a simulação do exercício anterior, mas atualize separadamente os componentes x e y da localização e da velocidade. Em cada iteração, plote a localização da bola de canhão na exibição gráfica como um pequeno círculo. Repita até que a bola de canhão alcance a terra novamente.

Esse tipo de problema tem importância histórica. Os primeiros computadores eram projetados para executar apenas esses cálculos balísticos, levando em conta a diminuição da gravidade para projéteis de longo alcance e a velocidade do vento.

★G **Exercício P6.15.** Escreva uma aplicação gráfica que exibe um tabuleiro de damas com 64 quadrados, alternando branco e preto.

★★G **Exercício P6.16.** Escreva uma aplicação gráfica que pede para um usuário inserir um número `n` e que desenhe `n` círculos com diâmetro e localização aleatórios. Os círculos devem estar completamente contidos dentro da janela.

★★★G **Exercício P6.17.** Escreva uma aplicação gráfica que desenhe uma espiral, como esta:



★★G **Exercício P6.18.** É fácil e divertido desenhar gráficos de curvas com a biblioteca de elementos gráficos de Java. Simplesmente desenhe 100 segmentos de linha unindo os pontos $(x, f(x))$ e $(x + d, f(x + d))$, onde x varia de x_{min} a x_{max} e $d = (x_{max} - x_{min}) / 100$. Desenhe a curva $f(x) = 0.00005x^3 - 0.03x^2 + 4x + 200$, onde x também varia de 0 a 400.

★★★G **Exercício P6.19.** Desenhe uma figura do trevo de quatro folhas cuja equação em coordenadas polares é $r = \cos(2\theta)$. Faça que θ vá de 0 a 2π em 100 passos. A cada passo, calcule `r` e então calcule as coordenadas (x, y) das coordenadas polares utilizando a fórmula

$$x = r\cos\theta, y = r\sin\theta$$

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 6.1.** *Índice de legibilidade de Flesch.* O índice a seguir [4] foi criado por Flesch como uma ferramenta para estimar a legibilidade de um documento sem a necessidade de uma análise lingüística.

- Conte todas as palavras no arquivo. Uma *palavra* é qualquer seqüência de caracteres delimitada por espaços em branco, seja ou não uma palavra inglesa real.
- Conte todas as sílabas em cada palavra. Para tornar isso simples, utilize estas regras: cada *grupo* de vogais adjacentes (a, e, i, o, u, y) é contado como uma sílaba (por exemplo, o "ea" em "real" contribui com uma sílaba, mas o "e . . . a" em "regal" é contado como duas sílabas). Mas um "e" no final de uma palavra não é contado como uma sílaba. Além disso, cada palavra tem pelo menos uma sílaba, mesmo se as regras anteriores fornecerem uma contagem de 0.
- Conte todas as frases. Uma frase termina por um ponto, dois-pontos, ponto-e-vírgula, ponto de interrogação ou sinal de exclamação.
- O índice é calculado por

$$\begin{aligned}\text{Index} &= 206.835 \\ &\quad - 84.6 \times (\text{Número de sílabas/Número de palavras}) \\ &\quad - 1.015 \times (\text{Número de palavras/Número de sentenças})\end{aligned}$$

  arredondado para o inteiro mais próximo.

Esse índice é um número, normalmente entre 0 e 100, indicando qual é o grau de dificuldade da leitura do texto. Alguns índices de exemplo para materiais aleatórios a partir de várias publicações são:

Revista em quadrinhos	95
Anúncios publicitários	82
Sports Illustrated	65
Time	57
New York Times	39
Apólice de seguro para automóveis	10
Código tributários dos EUA	–6

O propósito desse índice é forçar os autores a reescreverem seus textos até que o índice seja suficientemente alto. Isso é alcançado reduzindo o comprimento das frases e removendo palavras longas. Por exemplo, a frase

> O índice a seguir foi criado por Flesch como uma ferramenta simples para estimar a legibilidade de um documento sem a necessidade de uma análise lingüística.

pode ser reescrita como:

> Flesch criou um índice para verificar se um texto é fácil de ler. Para calcular o índice, você não precisa examinar o significado das palavras.

O livro de Flesch [4] contém ótimos exemplos da tradução de regulamentos governamentais em "inglês simples".

Traduzidos em níveis educacionais, os índices são:

91–100	aluno da 4º série do ensino fundamental
81–90	aluno da 5º série do ensino fundamental
71–80	aluno da 6º série do ensino fundamental
66–70	aluno da 7º série do ensino fundamental
61–65	aluno da 8º série do ensino fundamental
51–60	Aluno do ensino médio
31–50	Universitário incompleto
0–30	Universitário completo
Menor que 0	Bacharel em direito

Seu programa deve ler um arquivo de texto, calcular o índice de legibilidade e imprimir o nível educacional equivalente. Utilize as classes `Word` e `Document`.

★★★ **Projeto 6.2.** *O jogo de Nim.* Esse é um jogo famoso com muitas variantes. Consideraremos a variante a seguir, que tem uma interessante estratégia de vitória. Dois jogadores alternadamente retiram bolinhas de gude de uma pilha. Em cada movimento, um jogador escolhe quantas bolinhas de gude quer pegar. O jogador deve pegar pelo menos uma, mas no máximo metade das bolinhas de gude. Então o outro jogador faz uma jogada. O jogador que pegar a última bolinha de gude perde.

Escreva um programa no qual o computador joga contra um oponente humano. Gere um inteiro aleatório entre 10 e 100 para definir o tamanho inicial da pilha. Gere um inteiro aleatório entre 0 e 1 para decidir quem faz a primeira jogada. Gere um inteiro aleatório entre 0 e 1 para decidir se o computador joga no modo *inteligente* ou no modo *burro*. No modo burro, o computador simplesmente escolhe um valor aleatório válido (entre 1 e $n/2$) a partir da pilha sempre que for sua vez de jogar. No modo inteligente, o computador retira um número suficiente de bolinhas de gude para tornar o tamanho da pilha uma potência de dois menos 1 – isto é, 3, 7, 15, 31 ou 63. Esse valor sempre é um movimento válido, exceto se o tamanho atual da pilha for um a menos que uma potência de 2. Nesse caso, o computador faz uma jogada aleatória válida.

Observe que o computador não pode ser derrotado no modo inteligente quando for a vez dele de jogar, a menos que o tamanho da pilha seja 15, 31 ou 63. Naturalmente, um jogador humano que faz a primeira jogada e conhece a estratégia vitoriosa pode ganhar do computador.

Ao implementar esse programa, certifique-se de utilizar as classes `Pile`, `Player` e `Game`. Um jogador pode ser burro, inteligente ou humano. (Objetos `Player` humanos solicitam uma entrada.)

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

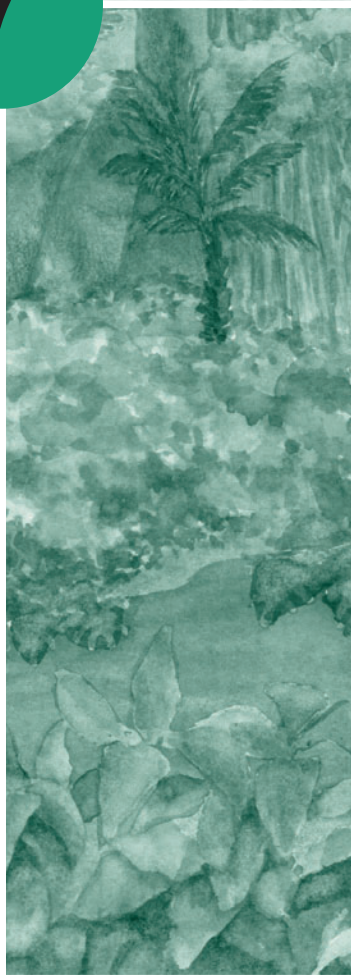
1. Nunca.
2. O método `waitForBalance` nunca retornaria devido a um laço infinito.
3. `int i = 1;`

```
while (i <= n)
{
   double interest = balance * rate / 100;
   balance = balance + interest;
   i++;
}
```

4. 11 vezes.
5. Altere o laço interno para `for (int j = 1; j <= width; j++)`.
6. 20.
7. Porque não sabemos se a próxima entrada é um número ou a letra `Q`.
8. Não. Se *todos* os valores de entrada forem negativos, o valor máximo também será negativo. Mas o campo `maximum` é inicializado com 0. Com essa simplificação, o valor máximo seria falsamente calculado como 0.
9. `int n = generator.nextInt(2); //` 0 = cara, 1 = coroa
10. O programa chama `Math.toRadians(angle)` repetidamente. Você simplesmente poderia chamar `Math.toRadians(180)` para calcular π.
11. Você deve usar um comando "step over" porque você não está interessado em depurar o funcionamento interno do método `println`.
12. Você deve configurar um ponto de interrupção. Percorrer laços pode ser entediante.
13. O programador entendeu erroneamente o segundo parâmetro do método substring – é o índice do primeiro caractere que não deve ser incluído na substring.
14. O segundo erro foi causado por falha em redefinir `insideVowelGroup` para falso no final de um grupo de vogais. Ele foi detectado rastreando o laço e percebendo que o laço não entrou na instrução condicional que incrementa a contagem de vogais.

Capítulo 7

Arrays e Listas de Arrays



Objetivos do capítulo

- Familiarizar-se com o uso de arrays e listas de arrays
- Aprender sobre classes empacotadoras, auto-empacotamento e o laço `for` aprimorado
- Analisar algoritmos comuns de manipulação arrays
- Aprender a utilizar arrays bidimensionais
- Entender quando escolher arrays e listas de arrays nos seus programas
- Implementar arrays parcialmente preenchidos
- T Entender o conceito do teste de regressão

Para processar grandes volumes de dados, você precisa coletar valores em uma estrutura de dados. As estruturas de dados mais comumente utilizadas em Java são arrays e listas de arrays. Neste capítulo, você aprenderá a construir arrays e listas de arrays, preenchê-los com valores e acessar os valores armazenados. Introduzimos o laço `for` aprimorado, uma instrução conveniente para processar todos os elementos de uma coleção. Veremos como utilizar o laço `for` aprimorado, bem como os laços convencionais, para implementar algoritmos comuns de manipulação de arrays. O capítulo conclui com uma seção técnica sobre como copiar valores de array.

Conteúdo do Capítulo

7.1 Arrays 272
Sintaxe 7.1: Construção de array 274
Sintaxe 7.2: Acesso ao elemento do array 275
Erro comum 7.1: Erros de limite 275
Erro comum 7.2: Arrays não-inicializados 276
Tópico avançado 7.1: Inicialização de array 276

7.2 Listas de arrays 276
Erro comum 7.3: Comprimento e tamanho 281
Dica de qualidade 7.1: Prefira listas de arrays parametrizadas ⊕

7.3 Empacotadores e auto-empacotamento 281

7.4 O laço for aprimorado 283
Sintaxe 7.3: O laço "for each" 285

7.5 Algoritmos simples para arrays 285

7.6 Arrays bidimensionais 290
Como fazer 7.1: Trabalhando com arrays e listas de arrays 293
Tópico avançado 7.2: Arrays bidimensionais com comprimentos de linha variáveis ⊕
Tópico avançado 7.3: Arrays multidimensionais ⊕

7.7 Copiando arrays 294
Erro comum 7.4: Subestimando o tamanho de um conjunto de dados 298
Dica de qualidade 7.2: Transforme arrays paralelos em arrays de objetos 298
Tópico avançado 7.4: Arrays parcialmente preenchidos ⊕
Tópico avançado 7.5: Métodos com um número variável de parâmetros ⊕
Fato aleatório 7.1: Um worm no início da Internet ⊕

7.8T Testes de regressão 300
Dica de produtividade 7.1: Arquivos batch e scripts de shell ⊕
Fato aleatório 7.2: Os incidentes do Therac-25 ⊕

7.1 Arrays

Em muitos programas, você precisa manipular coleções de valores relacionados. Seria complicado utilizar uma seqüência de variáveis como `data1`, `data2`, `data3`, . . . e assim por diante. A construção array fornece uma maneira mais adequada de armazenar uma coleção de valores.

Um array é uma seqüência de valores do mesmo tipo.

Um array é uma seqüência de valores do mesmo tipo. Por exemplo, eis como construir um array de 10 números de ponto flutuante:

```
new double[10]
```

O número de elementos (aqui, 10) é chamado de comprimento do array.

O operador `new` meramente constrói o array. É recomendável armazenar uma referência ao array em uma variável para que você possa acessá-lo mais tarde.

O tipo de uma variável de array é o tipo do elemento seguido por `[]`. Neste exemplo, o tipo é `double[]`, porque o tipo do elemento é `double`. Eis a declaração de uma variável de array:

```
double[] data = new double[10];
```

Isto é, `data` é uma referência a um array de números de ponto flutuante. Ele é inicializado com um array de 10 números (veja Figura 1).

Você também pode formar arrays de objetos, por exemplo

```
BankAccount[] accounts = new BankAccount[10];
```

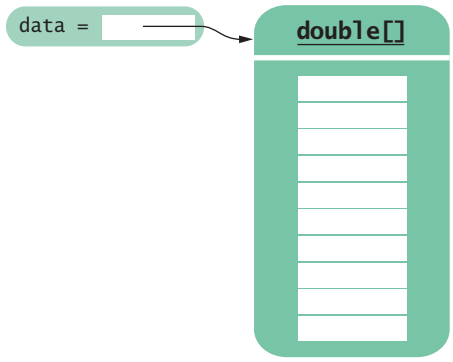


Figura 1 Uma referência a um array e um array.

Quando um array é criado pela primeira vez, todos os valores são inicializados com 0 (para um array de números como `int[]` ou `double[]`), `false` (para um array `boolean[]`) ou `null` (para um array de referências a objetos).

Cada elemento no array é especificado por um índice do tipo inteiro que é colocado entre colchetes (`[]`). Por exemplo, a expressão

```
data[4]
```

denota o elemento do array `data` com índice 4.

Você pode armazenar um valor em um local com uma instrução de atribuição, como esta:

```
data[2] = 29.95;
```

Você acessa elementos do array com um índice do tipo inteiro, utilizando a notação `a[i]`.

Agora a posição com o índice 2 de `data` é preenchida com o valor 29.95 (veja Figura 2).

Para ler o valor dos dados no índice 2, simplesmente utilize a expressão `data[2]` como você faria com qualquer variável do tipo `double`:

```
System.out.println("The value of this data item is "
      + data[2]);
```

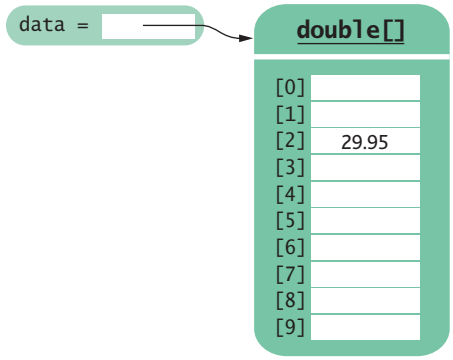


Figura 2 Armazenando um valor em um array.

Se examinar cuidadosamente a Figura 2, perceberá que os valores de índice iniciam em 0. Isto é,

```
data[0] é o primeiro elemento
data[1] é o segundo elemento
data[2] é o terceiro elemento
```

e assim por diante. Essa convenção pode ser uma fonte de problemas para o iniciante, portanto você deve prestar bastante atenção aos valores de índice. Em particular, o *último* elemento no array tem um índice "*um menor que*" o comprimento do array. Por exemplo, `data` refere-se a um array com comprimento 10. O último elemento é `data[9]`.

Valores de índice de um array variam de 0 a `length` – 1. Acessar um elemento inexistente resulta em um erro de limite.

Se tentar acessar um elemento que não existe, uma exceção será lançada. Por exemplo, a instrução

```
data[10] = 29.95; // ERRO
```

é um erro de limite.

Utilize o campo `length` para localizar o número de elementos em um array.

Para evitar erros de limite, você vai querer saber quantos elementos estão em um array. O campo `length` retorna o número de elementos: `data.length` é o comprimento do array `data`. Observe que não há nenhum parêntese depois de `length` – pois se trata de uma variável de instância do objeto array, não um método. Entretanto, você não pode atribuir um novo valor a essa variável de instância. Em outras palavras, `length` é uma variável de instância `final public`. Isso é uma anomalia. Normalmente, programadores Java utilizam um método para pesquisar as propriedades de um objeto. Nesse caso, você só precisa lembrar-se de omitir os parênteses.

O código a seguir assegura que você só acesse o array quando a variável de índice `i` está dentro dos limites válidos:

```
if (0 <= i && i < data.length) data[i] = value;
```

Os arrays apresentam uma limitação significativa: *o comprimento é fixo*. Se iniciar com um array de 10 elementos e mais tarde decidir que precisa adicionar outros elementos, você precisará criar um novo array e copiar todos os valores do array existente para o novo array. Discutiremos esse processo em detalhes na Seção 7.7.

SINTAXE 7.1 Construção de array

`new` *nomeDoTipo*[*comprimento*]

Exemplo:

```
new double[10]
```

Objetivo:

Construir um array com um dado número de elementos

SINTAXE 7.2 Acesso ao elemento do array

referênciaDeArray[*índice*]

Exemplo:

`data[2]`

Objetivo:

Acessar um elemento em um array

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Quais elementos o array `data` contém depois das seguintes instruções?

```
double[] data = new double[10];
for (int i = 0; i < data.length; i++) data[i] = i * i;
```

2. O que os segmentos de programa a seguir imprimem? Ou, se houver um erro, descreva esse erro e especifique se ele é detectado em tempo de compilação ou em tempo de execução.

   a.
```
double[] a = new double[10];
System.out.println(a[0]);
```

   b.
```
double[] b = new double[10];
System.out.println(b[10]);
```

   c.
```
double[] c;
System.out.println(c[0]);
```

ERRO COMUM 7.1

Erros de limite

O erro de array mais comum é tentar acessar uma posição inexistente.

```
double[] data = new double[10];
data[10] = 29.95;
// Erro – só há elementos com valores de índice 0 . . . 9
```

Quando o programa executa, um índice fora dos limites gera uma exceção e termina o programa.

Este é um excelente aprimoramento em relação a linguagens como o C e o C++. Nessas linguagens não há nenhuma mensagem de erro; em vez disso, o programa silenciosamente (ou não tão silenciosamente) irá corromper a posição da memória que está a 10 elementos do início do array. Às vezes, essa corrupção passa despercebida, mas outras vezes, o programa se comportará de maneira estranha ou sofrerá uma morte horrível muitas instruções mais tarde. Estes são problemas sérios que tornam programas C e C++ difíceis de depurar.

Erro comum 7.2

Arrays não-inicializados

Um erro comum é alocar uma referência de array, mas não um array real.

```
double[] data;
data[0] = 29.95; // Erro - data não inicializado
```

Variáveis de array funcionam exatamente como variáveis de objeto – elas são apenas referências ao array real. Para construir o array real, você deve utilizar o operador `new`:

```
double[] data = new double[10];
```

Tópico avançado 7.1

Inicialização de array

Você pode inicializar um array alocando-o e então preenchendo cada entrada:

```
int[] primes = new int[5];
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
primes[4] = 11;
```

Mas se você já conhece todos os elementos que quer colocar no array, há uma maneira mais fácil. Liste todos os elementos que você quer incluir no array, incluídos entre chaves e separados por vírgulas:

```
int[] primes = { 2, 3, 5, 7, 11 };
```

O compilador Java conta quantos elementos você quer colocar no array, aloca um array do tamanho correto e preenche-o com os elementos que você especifica.

Se quiser construir um array e passá-lo para um método que espera um parâmetro de array, você pode inicializar um array anônimo desta maneira:

```
new int[] { 2, 3, 5, 7, 11 }
```

7.2 Listas de arrays

A classe `ArrayList` gerencia uma seqüência de objetos.

Arrays são uma construção bastante primitiva. Nesta seção, introduziremos a classe `ArrayList` que permite colecionar objetos, tal como um array faz. Listas de arrays têm duas conveniências significativas:

- Listas de arrays podem aumentar e diminuir de tamanho conforme necessário
- A classe `ArrayList` fornece métodos para muitas tarefas comuns, por exemplo, inserir e remover elementos

Vamos definir uma lista de arrays de contas bancárias e preenchê-la com objetos. (A classe `BankAccount` foi aprimorada a partir da versão no Capítulo 3. Cada conta bancária tem um número de conta.)

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

A classe `ArrayList` é uma classe genérica: `ArrayList<T>` coleciona objetos do tipo `T`.

O tipo `ArrayList<BankAccount>` denota uma lista de arrays de contas bancárias. Os colchetes angulares em torno do tipo `BankAccount` informam que `BankAccount` é um *parâmetro de tipo*. Você pode substituir `BankAccount` por qualquer outra classe e obter um tipo diferente de lista de arrays. Por essa razão, `ArrayList` é chamado de classe *genérica*. Veremos outros detalhes das classes genéricas no Capítulo 17. Por enquanto, simplesmente utilize uma `ArrayList<T>` sempre que quiser coletar objetos do tipo `T`. Mas tenha em mente que você não pode utilizar tipos primitivos como parâmetros de tipo – não há nenhum `ArrayList<int>` ou `ArrayList<double>`.

Ao construir um objeto `ArrayList`, ele tem um tamanho 0. Utilize o método `add` para adicionar um objeto ao fim da lista de arrays. O tamanho aumenta depois de cada chamada a `add`. O método `size` fornece o tamanho atual da lista de arrays.

Para remover objetos da lista de arrays, utilize o método `get`, não o operador `[ ]`. Como ocorre com arrays, valores de índice iniciam em 0. Por exemplo, `accounts.get(2)` recupera a conta com índice 2, o terceiro elemento na lista de arrays:

```
BankAccount anAccount = accounts.get(2);
```

Como ocorre com arrays, é um erro acessar um elemento inexistente. O erro de limite mais comum é utilizar isto:

```
int i = accounts.size();
anAccount = accounts.get(i);  // Erro
```

O último índice válido é `accounts.size() - 1`.

Para configurar um elemento da lista de arrays como um novo valor, utilize o método `set`.

```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

Essa chamada configura a posição 2 da lista de arrays `accounts` com `anAccount`, sobrescrevendo qualquer valor que estivesse ali anteriormente.

O método `set` só pode sobrescrever valores existentes. Ele é diferente do método `add`, que adiciona um novo objeto ao final da lista de arrays.

Você também pode inserir um objeto no meio de uma lista de arrays. A chamada `accounts.add(i, a)` adiciona o objeto `a` na posição `i` e move todos os elementos por uma posição, do elemento atual na posição `i` ao último elemento na lista de arrays. Depois de cada chamada ao método `add`, o tamanho da lista de arrays aumenta de 1 (veja Figura 3).

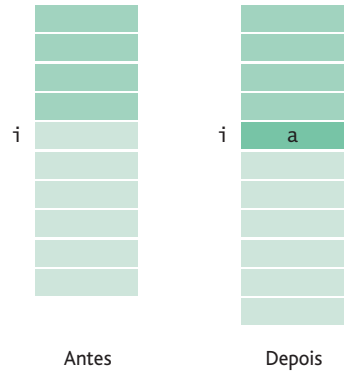


Figura 3 Adicionando um elemento no meio de uma lista de arrays.

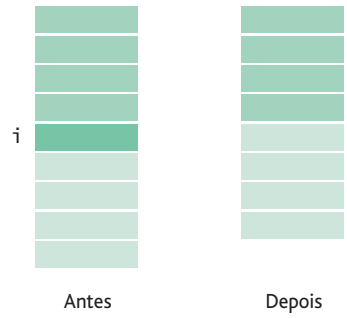


Figura 4 Removendo um elemento do meio de uma lista de arrays.

Inversamente, a chamada `accounts.remove(i)` remove o elemento na posição `i`, move todos os elementos depois do elemento removido para baixo por uma posição e reduz o tamanho da lista de arrays de 1 (veja Figura 4).

O programa a seguir demonstra os métodos da classe `ArrayList`. Observe que você importa a classe genérica `java.util.ArrayList` sem o parâmetro de tipo.

ch07/arraylist/ArrayListTester.java

```
import java.util.ArrayList;

/**
   Esse programa testa a classe ArrayList
*/
public class ArrayListTester
{
```

```
8     public static void main(String[] args)
9     {
10       ArrayList<BankAccount> accounts
11             = new ArrayList<BankAccount>();
12       accounts.add(new BankAccount(1001));
13       accounts.add(new BankAccount(1015));
14       accounts.add(new BankAccount(1729));
15       accounts.add(1, new BankAccount(1008));
16       accounts.remove(0);
17
18       System.out.println("Size: " + accounts.size());
19       System.out.println("Expected: 3");
20       BankAccount first = accounts.get(0);
21       System.out.println("First account number: "
22             + first.getAccountNumber());
23       System.out.println("Expected: 1008");
24       BankAccount last = accounts.get(accounts.size() - 1);
25       System.out.println("Last account number: "
26             + last.getAccountNumber());
27       System.out.println("Expected: 1729");
28    }
29 }
```

ch07/arraylist/BankAccount.java

```
1  /**
2     Uma conta bancária tem um saldo que pode ser alterado por
3     depósitos e saques.
4  */
5  public class BankAccount
6  {
7     /**
8        Cria uma conta bancária com saldo zero.
9        @param anAccountNumber número dessa conta
10    */
11    public BankAccount(int anAccountNumber)
12    {
13       accountNumber = anAccountNumber;
14       balance = 0;
15    }
16
17    /**
18       Constrói uma conta bancária com um saldo especificado.
19       @param anAccountNumber número dessa conta
20       @param initialBalance saldo inicial
21    */
22    public BankAccount(int anAccountNumber, double initialBalance)
23    {
24       accountNumber = anAccountNumber;
25       balance = initialBalance;
26    }
27
```

```
28      /**
29         Obtém o número dessa conta bancária.
30         @return número da conta
31      */
32      public int getAccountNumber()
33      {
34         return accountNumber;
35      }
36
37      /**
38         Deposita dinheiro na conta bancária.
39         @param amount valor a depositar
40      */
41      public void deposit(double amount)
42      {
43         double newBalance = balance + amount;
44         balance = newBalance;
45      }
46
47      /**
48         Retira dinheiro da conta bancária.
49         @param amount valor a retirar
50      */
51      public void withdraw(double amount)
52      {
53         double newBalance = balance - amount;
54         balance = newBalance;
55      }
56
57      /**
58         Obtém o saldo atual da conta bancária.
59         @return saldo atual
60      */
61      public double getBalance()
62      {
63         return balance;
64      }
65
66      private int accountNumber;
67      private double balance;
68   }
```

Saída

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Como você constrói um array de 10 strings? E uma lista de arrays de strings?
4. Qual é o conteúdo de `names` depois destas instruções?

```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

ERRO COMUM 7.3

Comprimento e tamanho

Infelizmente, a sintaxe de Java para determinar o número de elementos em um array, em uma lista de arrays e em uma string é bem inconsistente. É um erro comum confundi-los. Você simplesmente precisa lembrar-se da sintaxe correta para cada tipo de dados.

Tipo de dados	Número de elementos
Array	`a.length`
Lista de arrays	`a.size()`
`String`	`a.length()`

DICA DE QUALIDADE 7.1

Prefira listas de arrays parametrizadas

A Dica de Qualidade 7.1 recomenda que você sempre utilize um parâmetro de tipo com uma `ArrayList`. Por exemplo, você deve utilizar `ArrayList<BankAccount>`, não o `ArrayList` não-tipado, ainda que o último seja compilado para compatibilidade com versões mais antigas de Java.

7.3 Empacotadores e auto-empacotamento

Para tratar valores de tipos primitivos como objetos, você deve utilizar classes empacotadoras (wrappers).

Como números não são objetos em Java, você não pode inseri-los diretamente nas listas de arrays. Por exemplo, você não pode formar um `ArrayList<double>`. Para armazenar seqüências de números em uma lista de arrays, primeiro você precisa transformá-los em objetos utilizando classes empacotadoras.

Há classes empacotadoras para os oito tipos primitivos:

Tipo primitivo	Classe empacotadora
`byte`	`Byte`
`boolean`	`Boolean`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

Observe que os nomes das classes empacotadoras iniciam com letras maiúsculas e também que duas delas diferem dos nomes dos tipos primitivos correspondentes: `Integer` e `Character`.

Cada objeto de classe empacotadora contém um valor do tipo primitivo correspondente. Por exemplo, um objeto da classe `Double` contém um valor do tipo `double` (veja Figura 5).

Objetos empacotadores podem ser utilizados em qualquer lugar em que objetos são requeridos no lugar de valores de tipo primitivo. Por exemplo, você pode coletar uma seqüência de números de ponto flutuante em um `ArrayList<Double>`.

A partir de Java 5.0, a conversão entre tipos primitivos e classes empacotadoras correspondentes é automática. Esse processo é chamado de auto-empacotamento ou *auto-boxing* (embora *auto-wrapping* fosse mais consistente).

Por exemplo, se atribuir um número a um objeto `Double`, o número é automaticamente "colocado em uma caixa [box]", a saber, um objeto empacotador.

```
Double d = 29.95; // auto-empacotamento; o mesmo que Double d = new Double(29.95);
```

Se utilizar uma versão mais antiga de Java, você mesmo precisará fornecer o construtor.

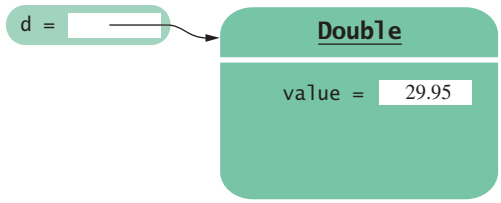


Figura 5 Um objeto de uma classe empacotadora.

Inversamente, a partir de Java 5.0, objetos de classes empacotadoras são automaticamente "desempacotados" para tipos primitivos.

```
double x = d; // auto-desempacotamento; o mesmo que double x = d.doubleValue();
```

Com versões mais antigas, você precisa chamar um método como `doubleValue`, `intValue` ou `booleanValue` para fazer o desempacotamento.

O auto-empacotamento funciona até mesmo em expressões aritméticas. Por exemplo, a instrução

```
Double e = d + 1;
```

é perfeitamente válida. Significa:

- Desempacote `d` e o converta em um `double`.
- Some 1.
- Desempacote o resultado e o converta em um novo `Double`.
- Armazene uma referência ao objeto da classe empacotadora recém-criado em `e`.

Se você usa Java 5.0, ou uma versão mais atual, listas de arrays de números são simples e diretas. Basta lembrar-se de utilizar o tipo empacotador quando você declarar a lista de arrays e então usar o auto-empacotamento.

```
ArrayList<Double> data = new ArrayList<Double>();
data.add(29.95);
double x = data.get(0);
```

Com as versões mais antigas de Java, utilizar classes empacotadoras para armazenar números em uma lista de arrays é bastante trabalhoso porque você precisa empacotar e desempacotar manualmente os números.

Independentemente da versão de Java que você usa, você precisa saber que armazenar números empacotados é bem ineficiente. O uso de empacotadores é aceitável para pequenas listas de arrays, mas você deve utilizar arrays para seqüências longas de números ou caracteres.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

5. Qual é a diferença entre os tipos `double` e `Double`?
6. Suponha que `data` seja uma `ArrayList<Double>` de tamanho > 0. Como você incrementa o elemento com o índice 0?

7.4 O laço for aprimorado

Java 5.0 introduz um atalho muito conveniente para um tipo comum de laço. Freqüentemente, você precisa iterar por uma seqüência de elementos – como os elementos de um array ou de uma lista de arrays. O laço `for` aprimorado torna esse processo particularmente fácil de programar.

O laço `for` aprimorado percorre todos os elementos de uma coleção.

Suponha que você queira somar todos os valores dos dados em um array `data`. Eis como utilizar o laço `for` aprimorado para executar essa tarefa.

```
double[] data = . . .;
double sum = 0;
for (double e : data)
{
   sum = sum + e;
}
```

O corpo do laço é executado para cada elemento no array `data`. No início de cada iteração do laço, o próximo elemento é atribuído à variável `e`. O corpo do laço é então executado. Você deve ler esse laço como "para cada `e` em `data`".

Você poderia perguntar por que Java não permite escrever "`for each (e in data)`". Sem dúvida, seria mais conveniente e os projetistas da linguagem Java consideraram seriamente fazer isso. Mas, a construção "*for each*" foi adicionada à Java bem depois da sua distribuição inicial. Novas palavras-chave `each` e `in` deveriam ser adicionadas à linguagem, assim, programas mais antigos que utilizavam esses identificadores como nomes de variáveis ou nomes de método (como `System.in`) não mais seriam compilados corretamente.

Você não é obrigado a utilizar a construção "*for each*" para fazer um laço para percorrer todos os elementos em um array. Você pode implementar o mesmo laço com um laço `for` simples e direto e uma variável de índice explícita:

```
double[] data = . . .;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
   double e = data[i];
   sum = sum + e;
}
```

Observe uma diferença importante entre o laço "*for each*" e o laço `for` normal. No laço "*for each*" a *variável elemento* `e` recebe valores `data[0]`, `data[1]` e assim por diante. No laço `for` normal, a *variável índice* `i` recebe os valores 0, 1, etc.

Você também pode utilizar o laço `for` aprimorado para acessar todos os elementos de uma lista de arrays. Por exemplo, o laço a seguir calcula o valor total de todas as contas:

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
   sum = sum + a.getBalance();
}
```

Esse laço é equivalente ao laço `for` normal a seguir:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   sum = sum + a.getBalance();
}
```

O laço "*for each*" tem um propósito bem específico: percorrer os elementos de uma coleção, do início ao fim. Às vezes você não quer começar do início, ou talvez precise percorrer a coleção de trás para frente. Nessas situações, não hesite em utilizar um laço `for` normal.

SINTAXE 7.3 O laço "for each"

```
for (Tipo variável : coleção)
   instrução
```

Exemplo:

```
for (double e : data)
   sum = sum + e;
```

Objetivo:

Executar um laço para cada elemento na coleção. Em cada iteração, a variável recebe o próximo elemento da coleção. A instrução é então executada.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

7. Escreva um laço "*for each*" que imprime todos os elementos no array `data`.
8. Por que o laço "*for each*" não é um atalho apropriado para o laço `for` normal a seguir?

```
for (int i = 0; i < data.length; i++) data[i] = i * i;
```

7.5 Algoritmos simples para arrays

7.5.1 Contando correspondências

Para contar os valores em uma lista de arrays, verifique todos os elementos e conte as correspondências até alcançar o fim da lista de arrays.

Suponha que você queira localizar quantas contas de certo tipo você tem. Você deve então pesquisar toda a coleção e incrementar um contador toda vez que encontrar uma correspondência. Aqui contamos o número de contas cujo saldo seja pelo menos igual a um dado limiar:

```
public class Bank
{
   public int count(double atLeast)
   {
      int matches = 0;
      for (BankAccount a : accounts)
      {
         if (a.getBalance() >= atLeast) matches++;
            // Encontrou uma correspondência
```

```
      }
      return matches;
   }
   . . .
   private ArrayList<BankAccount> accounts;
}
```

7.5.2 Localizando um valor

Para localizar um valor em uma lista de arrays, verifique todos os elementos até encontrar uma correspondência.

Suponha que você queira saber se há uma conta bancária com um número de conta particular no seu banco. Simplesmente inspecione cada elemento até encontrar uma correspondência ou alcançar o fim da lista de arrays. Observe que talvez o laço não consiga localizar uma resposta, que é o caso quando nenhuma das contas corresponde. Esse processo de pesquisa é chamado pesquisa linear pela lista de arrays.

```
public class Bank
{
   public BankAccount find(int accountNumber)
   {
      for (BankAccount a : accounts)
      {
         if (a.getAccountNumber() == accountNumber) // Encontrou uma correspondência
            return a;
      }
      return null; // Nenhuma correspondência em toda a lista de arrays
   }
   . . .
}
```

Observe que o método retorna `null` se nenhuma correspondência for localizada.

7.5.3 Localizando o máximo ou o mínimo

Para calcular o valor máximo ou mínimo de uma lista de arrays, inicialize um candidato com o elemento inicial. Compare então o candidato aos elementos restantes e atualize-o se encontrar um valor maior ou menor.

Suponha que você queira localizar a conta com o maior saldo no banco. Mantenha um candidato para o valor máximo. Se encontrar um elemento com um valor maior, substitua o candidato por esse valor. Depois de alcançar o fim da lista de arrays, você encontrou o valor máximo.

Há apenas um problema. Ao acessar o começo do array, você ainda não tem um candidato para o valor máximo. Uma maneira de solucionar isso é configurar o candidato como o elemento inicial do array e começar a comparação com o próximo elemento.

```
BankAccount largestYet = accounts.get(0);
for (int i = 1; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   if (a.getBalance() > largestYet.getBalance())
      largestYet = a;
}
return largestYet;
```

Agora utilizamos um laço `for` explícito porque o laço não mais acessa todos os elementos – ele pula o elemento inicial.

Naturalmente, essa abordagem só funciona se houver pelo menos um elemento na lista de arrays. Não faz muito sentido procurar o maior elemento de uma coleção vazia. Podemos retornar `null` nesse caso:

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
. . .
```

Consulte nos Exercícios R7.5 e R7.6 pequenas modificações nesse algoritmo.

Para calcular o valor mínimo de um conjunto de dados, mantenha um candidato para o valor mínimo e o substitua sempre que você encontrar um valor *menor*. No fim da lista de arrays, você encontra o valor mínimo.

O programa de exemplo a seguir implementa uma classe `Bank` que armazena uma lista de arrays de contas bancárias. Os métodos da classe `Bank` utilizam os algoritmos discutidos nesta seção.

ch07/bank/Bank.java

```
1  import java.util.ArrayList;
2
3  /**
4     Esse banco contém uma coleção de contas bancárias.
5  */
6  public class Bank
7  {
8     /**
9        Constrói um banco sem nenhuma conta bancária.
10    */
11    public Bank()
12    {
13       accounts = new ArrayList<BankAccount>();
14    }
15
16    /**
17       Adiciona uma conta a esse banco.
18       @param a conta a adicionar
19    */
20    public void addAccount(BankAccount a)
21    {
22       accounts.add(a);
23    }
24
25    /**
26       Obtém a soma dos saldos de todas as contas nesse banco.
27       @return soma dos saldos
28    */
29    public double getTotalBalance()
30    {
31       double total = 0;
32       for (BankAccount a : accounts)
33       {
34          total = total + a.getBalance();
```

```
35         }
36         return total;
37      }
38
39      /**
40         Conta quantas contas bancárias tem saldo maior ou igual a
41         um dado valor.
42         @param atLeast saldo necessário para fazer a contagem de uma conta
43         @return número de contas com pelo menos o saldo especificado
44      */
45      public int count(double atLeast)
46      {
47         int matches = 0;
48         for (BankAccount a : accounts)
49         {
50            if (a.getBalance() >= atLeast) matches++; // Encontrou
                                                          correspondência
51         }
52         return matches;
53      }
54
55      /**
56         Localiza uma conta bancária com um dado número.
57         @param accountNumber número a localizar
58         @return conta com o número espcificado, ou null se não existir
59         uma conta assim
60      */
61      public BankAccount find(int accountNumber)
62      {
63         for (BankAccount a : accounts)
64         {
65            if (a.getAccountNumber() == accountNumber) // Encontrou
                                                            correspondência
66               return a;
67         }
68         return null; // Nenhuma correspondência em toda a lista de arrays
69      }
70
71      /**
72         Obtém a conta bancária com o maior saldo.
73         @return conta com o maior saldo, ou null se o
74         banco não tiver nenhuma conta
75      */
76      public BankAccount getMaximum()
77      {
78         if (accounts.size() == 0) return null;
79         BankAccount largestYet = accounts.get(0);
80         for (int i = 1; i < accounts.size(); i++)
81         {
82            BankAccount a = accounts.get(i);
83            if (a.getBalance() > largestYet.getBalance())
84               largestYet = a;
85         }
86         return largestYet;
87       }
88
```

```
89  private ArrayList<BankAccount> accounts;
90  }
```

ch07/bank/BankTester.java

```
1   /**
2      Este programa testa a classe Bank.
3   */
4   public class BankTester
5   {
6      public static void main(String[] args)
7      {
8         Bank firstBankOfJava = new Bank();
9         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
10        firstBankOfJava.addAccount(new BankAccount(1015, 10000));
11        firstBankOfJava.addAccount(new BankAccount(1729, 15000));
12
13        double threshold = 15000;
14        int c = firstBankOfJava.count(threshold);
15        System.out.println("Count: " + c);
16        System.out.println("Expected: 2");
17
18        int accountNumber = 1015;
19        BankAccount a = firstBankOfJava.find(accountNumber);
20        if (a == null)
21           System.out.println("No matching account");
22        else
23           System.out.println("Balance of matching account: "
24                 + a.getBalance());
25        System.out.println("Expected: 10000");
26
27        BankAccount max = firstBankOfJava.getMaximum();
28        System.out.println("Account with largest balance: "
29              + max.getAccountNumber());
30        System.out.println("Expected: 1001");
31     }
32  }
```

Saída

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

9. O que o método `find` faz se houver duas contas bancárias com um mesmo número de conta?
10. Seria possível utilizar um laço "*for each*" no método `getMaximum`?

7.6 Arrays bidimensionais

Arrays e listas de arrays podem armazenar seqüências lineares. Ocasionalmente você quer armazenar coleções que têm um leiaute bidimensional. O exemplo tradicional é o tabuleiro do jogo da velha (veja Figura 6).

Arrays bidimensionais formam um arranjo bidimensional tabular. Você acessa os elementos com um par de índices `a[i][j]`.

Esse arranjo, que consiste em linhas e colunas de valores, é chamado array bidimensional ou matriz. Ao construir um array bidimensional, você especifica quantas linhas e colunas você precisa. Neste caso, 3 linhas e 3 colunas:

```
final int ROWS = 3;
final int COLUMNS = 3;
String[][] board = new String[ROWS][COLUMNS];
```

Isso produz um array bidimensional com 9 elementos

```
board[0][0]   board[0][1]   board[0][2]
board[1][0]   board[1][1]   board[1][2]
board[2][0]   board[2][1]   board[2][2]
```

Para acessar um elemento particular, especifique dois subscritos em colchetes separados:

```
board[i][j] = "x";
```

Ao preencher ou pesquisar um array bidimensional, é comum utilizar dois laços aninhados. Por exemplo, este par de laços configura todos os elementos no array como espaços.

```
for (int i = 0; i < ROWS; i++)
   for (int j = 0; j < COLUMNS; j++)
      board[i][j] = " ";
```

Eis uma classe e um programa de teste para o jogo da velha. Essa classe não verifica se um usuário venceu o jogo. Isso você pode descobrir ao resolver o Exercício P7.10.

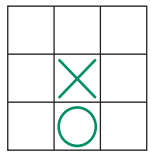


Figura 6
Tabuleiro de jogo da velha.

ch07/twodim/TicTacToe.java

```
1  /**
2     Um tabuleiro de jogo da velha de 3 x 3.
3  */
4  public class TicTacToe
5  {
6     /**
7        Constrói um tabuleiro vazio.
8     */
9     public TicTacToe()
10    {
```

```
11         board = new String[ROWS][COLUMNS];
12         // Preenche com espaços
13         for (int i = 0; i < ROWS; i++)
14            for (int j = 0; j < COLUMNS; j++)
15               board[i][j] = " ";
16      }
17
18      /**
19         Configura um campo no tabuleiro. O campo deve estar vazio.
20         @param i índice de linha
21         @param j índice de coluna
22         @param player jogador ("x" ou "o")
23      */
24      public void set(int i, int j, String player)
25      {
26         if (board[i][j].equals(" "))
27            board[i][j] = player;
28      }
29
30      /**
31         Cria uma representação de string do tabuleiro, como
32         |x o|
33         | x |
34         |  o|.
35         @return representação de string
36      */
37      public String toString()
38      {
39         String r = "";
40         for (int i = 0; i < ROWS; i++)
41         {
42            r = r + "|";
43            for (int j = 0; j < COLUMNS; j++)
44               r = r + board[i][j];
45            r = r + "|\n";
46         }
47         return r;
48      }
49
50      private String[][] board;
51      private static final int ROWS = 3;
52      private static final int COLUMNS = 3;
53   }
```

ch07/twodim/TicTacToeRunner.java

```
1   import java.util.Scanner;
2
3   /**
4      Esse programa executa um jogo da velha. Ele pede para
5      o usuário configurar as posições no tabuleiro e imprime o
6      resultado.
7   */
```

```
public class TicTacToeRunner
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      String player = "x";
      TicTacToe game = new TicTacToe();
      boolean done = false;
      while (!done)
      {
         System.out.print(game.toString());
         System.out.print(
               "Row for " + player + " (-1 to exit): ");
         int row = in.nextInt();
         if (row < 0) done = true;
         else
         {
            System.out.print("Column for " + player + ": ");
            int column = in.nextInt();
            game.set(row, column, player);
            if (player.equals("x"))
               player = "o";
            else
               player = "x";
         }
      }
   }
}
```

Saída

```
|   |
|   |
|   |
Row for x (-1 to exit): 1
Column for x: 2
|   |
|  x|
|   |
Row for o (-1 to exit): 0
Column for o: 0
|o  |
|  x|
|   |
Row for x (-1 to exit): -1
```

Autoverificação da aprendizagem

11. Como você declara e inicializa um array de inteiros de 4 por 4?
12. Como você conta o número de espaços no tabuleiro do jogo da velha?

COMO FAZER 7.1

Trabalhando com arrays e listas de arrays

Passo 1 Selecione a estrutura de dados apropriada.

Como regra geral, sua primeira escolha deve ser uma lista de arrays. Utilize um array apenas se desejar colecionar números (ou outros valores de tipo primitivo) e se for uma questão de eficiência, ou se você precisar de um array bidimensional.

Passo 2 Construa a lista de arrays ou o array e salve uma referência em uma variável.

Tanto para listas de arrays como para arrays, você precisa especificar o tipo de elemento. Para um array, você também precisa especificar o comprimento.

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
double[] balances = new double[n];
```

Passo 3 Adicione elementos.

Para uma lista de arrays, simplesmente chame o método `add`. Cada chamada adiciona um elemento ao final.

```
accounts.add(new BankAccount(1008));
accounts.add(new BankAccount(1729));
```

Para um array, utilize valores de índice para acessar os elementos.

```
balance[0] = 29.95;
balance[1] = 1000;
```

Passo 4 Processe os elementos.

O padrão mais comum de processamento envolve o acesso a todos os elementos na coleção. Utilize o laço "*for each*" para esse propósito:

```
for (BankAccount a : accounts)
   Faz algo com a
```

Se você não precisa examinar todos os elementos, utilize, em vez disso, um laço normal. Por exemplo, para pular o elemento inicial, você pode utilizar este laço.

```
for (int i = 1; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
      Faz algo com a
}
```

Para arrays, utilize `.length` em vez de `.size()` e `[i]` em vez de `.get(i)`.

TÓPICO AVANÇADO 7.2

Arrays bidimensionais com comprimentos de linha variáveis

O Tópico Avançado 7.2 discute os arrays bidimensionais nos quais linhas têm comprimentos diferentes.

TÓPICO AVANÇADO 7.3

Arrays multidimensionais

O Tópico avançado 7.3 discute arrays de três ou mais dimensões.

7.7 Copiando arrays

Uma variável do tipo array armazena uma referência ao array. Copiar a variável fornece uma segunda referência ao mesmo array.

Variáveis do tipo array funcionam exatamente como variáveis de objeto – elas armazenam uma referência ao array real. Se copiar a referência, você obterá uma outra referência ao mesmo array (veja Figura 7):

```
double[] data = new double[10];
. . .  // Preenche o array
double[] prices = data;
```

Utilize o método `clone` para copiar os elementos de um array.

Se quiser fazer uma cópia verdadeira de um array, chame o método `clone` (veja Figura 8).

```
double[] prices = (double[]) data.clone();
```

O método `clone` (que veremos mais adiante no Capítulo 10) tem o tipo de retorno `Object`. Você precisa fazer uma coerção (ou typecasting) do valor de retorno do método `clone` para o tipo apropriado de array como `double[]`.

Utilize o método `System.arraycopy` para copiar elementos de um array para outro.

Ocasionalmente, você precisará copiar elementos de um array para outro. Você pode utilizar o método `System.arraycopy` estático para esse propósito (veja Figura 9):

```
System.arraycopy(from, fromStart, to, toStart, count);
```

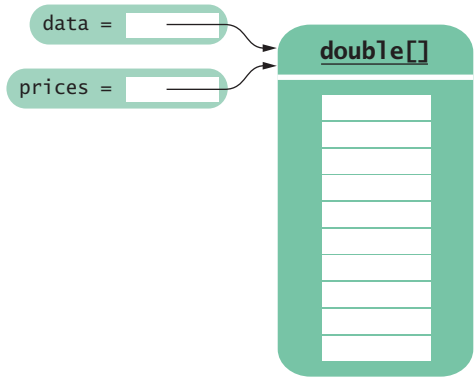


Figura 7 Duas referências ao mesmo array.

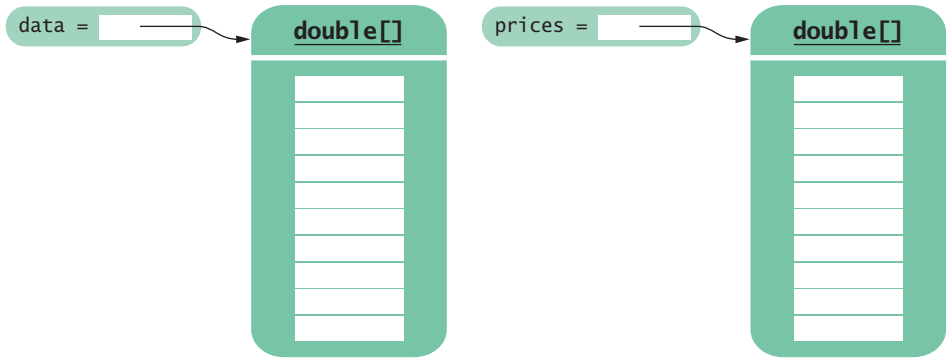


Figura 8 Clonando um array.

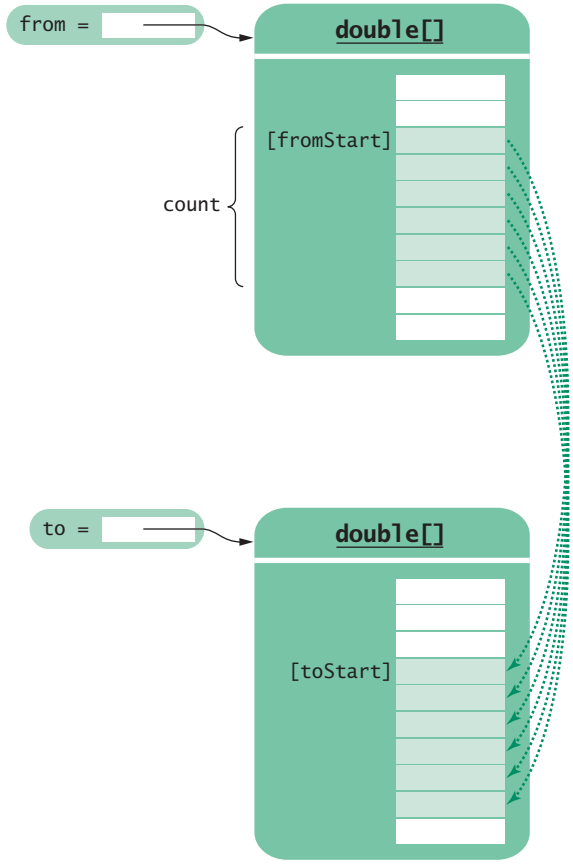


Figura 9 O método `System.arraycopy`.

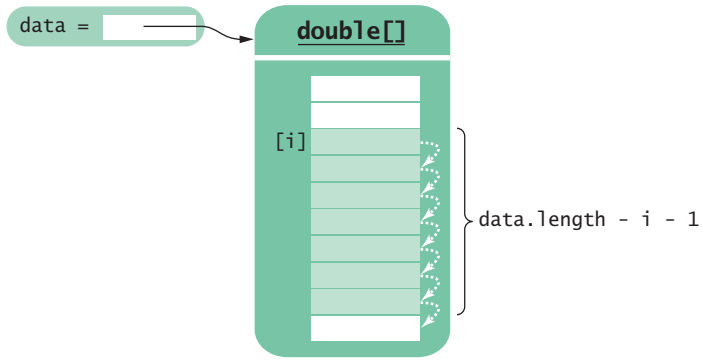


Figura 10 Inserindo um novo elemento em um array.

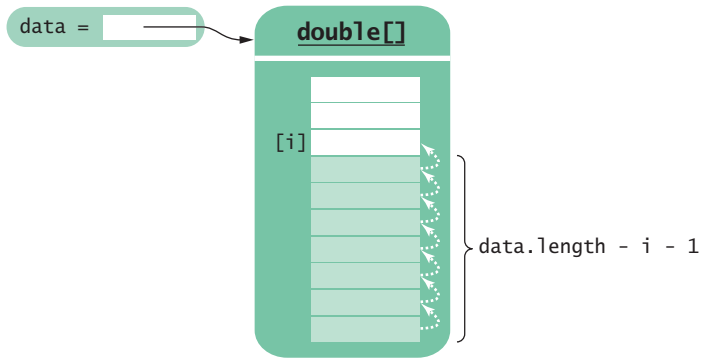


Figura 11 Removendo um elemento de um array.

Um dos usos do método `System.arraycopy` é adicionar ou remover elementos no meio de um array. Para adicionar um novo elemento na posição `i` em `data`, primeiro mova todos os elementos de `i` para uma posição acima. Insira então o novo valor.

```
System.arraycopy(data, i, data, i + 1, data.length - i - 1);
data[i] = x;
```

Observe que o último elemento do array é perdido (veja Figura 10).

Para remover o elemento da posição `i`, copie os elementos acima da posição para baixo (veja Figura 11).

```
System.arraycopy(data, i + 1, data, i, data.length - i - 1);
```

Um outro uso do `System.arraycopy` é aumentar e alocar mais espaço em um array. Siga estes passos:

- Crie um array novo e maior.
  `double[] newData = new double[2 * data.length];` ❶
- Copie todos os elementos para o novo array.
  `System.arraycopy(data, 0, newData, 0, data.length);` ❷

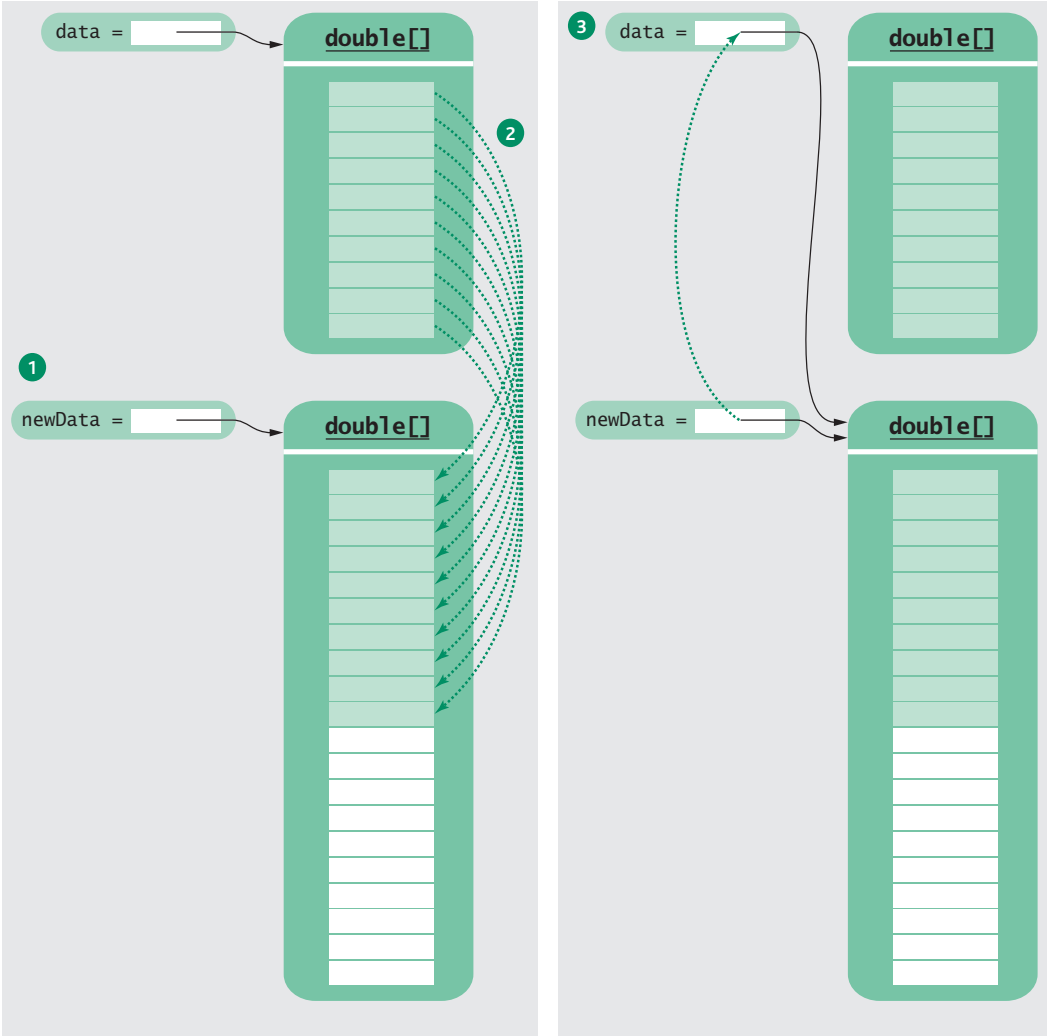


Figura 12 Expandindo um array.

- Armazene a referência ao novo array na variável de array.
  `data = newData;` 3

A Figura 12 mostra o processo.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

13. Como você adiciona ou remove elementos no meio de uma lista de arrays?
14. Por que dobramos o comprimento do array quando ele não tem mais espaço em vez de aumentá-lo de um elemento?

Erro comum 7.4

Subestimando o tamanho de um conjunto de dados

Programadores comumente subestimam o volume de dados de entrada que um usuário adicionará a um programa inocente. O problema mais comum causado por subestimar volume de dados de entrada é resultado do uso de arrays de tamanho fixo. Suponha que você escreva um programa para pesquisar texto em um arquivo. Você armazena cada linha em uma string e mantém um array de strings. Qual deve ser o tamanho do array? Seguramente ninguém desafiará seu programa com uma entrada que tenha mais de 100 linhas. Sério? Um aluno inteligente pode adicionar facilmente todo o livro *Alice no País das Maravilhas* ou *Guerra e Paz* (disponíveis na Internet). De repente, seu programa precisa gerenciar dezenas, centenas ou milhares de linhas. O que ele fará? Ele tratará a entrada? Gentilmente rejeitará a entrada em excesso? Irá travar e morrer?

Um famoso artigo [1] analisou como vários programas UNIX reagiam quando eram alimentados com conjuntos de dados grandes ou aleatórios. Infelizmente, cerca de 25% deles simplesmente não funcionou bem, travando ou parando sem uma mensagem de erro razoável. Por exemplo, em algumas versões do UNIX o programa de backup de fita `tar` não consegue tratar nomes de arquivo com mais de 100 caracteres, o que é uma limitação absurda. Muitas dessas falhas são causadas por recursos da linguagem C que, diferentemente de Java, torna difícil armazenar strings de tamanho arbitrário.

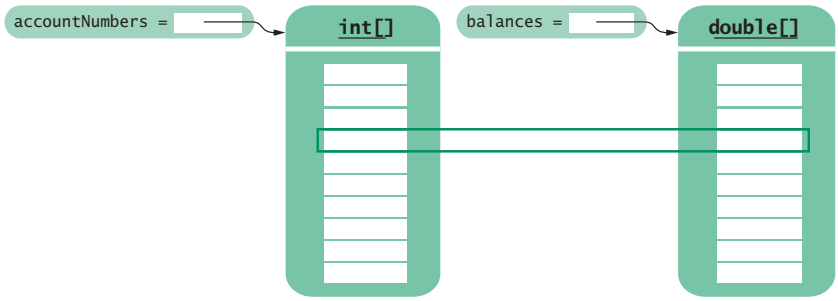
Dica de qualidade 7.2

Transforme arrays paralelos em arrays de objetos

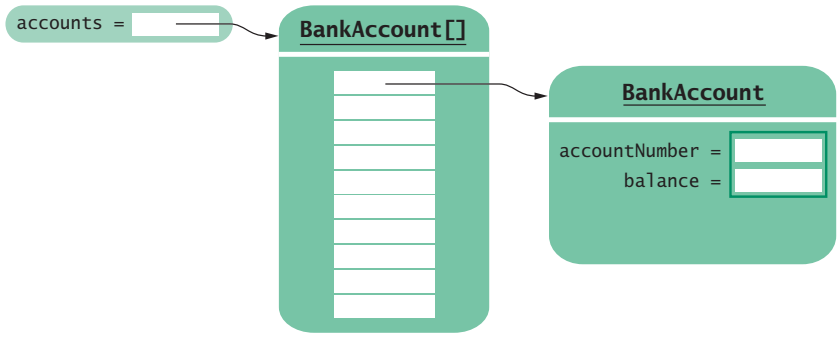
Programadores que conhecem arrays, mas que conhecem pouco de programação orientada a objetos, às vezes separam as informações em diferentes arrays. Eis um exemplo típico. Um programa precisa gerenciar dados do banco, que consistem em números de conta e saldos. Não armazene os números de conta e os saldos em arrays separados.

```
// Não faça isso
int[] accountNumbers;
double[] balances;
```

Arrays como esses são chamados arrays paralelos (ver Ilustração abaixo). A `i`-ésima fatia (`accountNumbers[i]` e `balances[i]`) contém dados que precisam ser processados conjuntamente.



Evite arrays paralelos



Reorganizando arrays paralelos em um array de objetos.

Se utilizar dois arrays que tenham o mesmo comprimento, pense se você não poderia substituí-los por um único array de um tipo de classe. Examine uma fatia e localize o conceito que ela representa. Transforme esse conceito em uma classe. No nosso exemplo, cada fatia contém um número de conta e um saldo, descrevendo uma conta bancária. Portanto, é uma simples questão de utilizar um único array de objetos

> Evite arrays paralelos transformando-os em arrays de objetos.

```
BankAccount[] accounts;
```

(Ver ilustração acima.) Ou, melhor ainda, utilize um `ArrayList<BankAccount>`.

Por que isso é benéfico? Pense à frente. Talvez seu programa seja alterado e você também precise armazenar o nome do proprietário da conta bancária. É só uma questão de atualizar a classe `BankAccount`. Poderia ser bem complicado adicionar um novo array e certificar-se de que todos os métodos que acessavam os dois arrays originais agora também acessam corretamente o terceiro.

TÓPICO AVANÇADO 7.4

Arrays parcialmente preenchidos

Diferentemente das listas de arrays, arrays têm um comprimento fixo. O Tópico Avançado 7.4 mostra como monitorar os elementos que realmente são utilizados em um array parcialmente preenchido.

TÓPICO AVANÇADO 7.5

Métodos com um número variável de parâmetros

O Tópico Avançado 7.5 demonstra como implementar um método que aceita um número variável de parâmetros e como recuperar os valores de parâmetro a partir de um array.

Fato aleatório 7.1

Um worm no início da internet

O Fato Aleatório 7.1 comenta a história do primeiro vírus sério da internet, distribuído por um aluno de pós-graduação da Universidade de Cornell. Esse vírus explorou uma vulnerabilidade de overrun de array presente na linguagem de programação C, mas não em Java.

7.8 Testes de regressão

Um conjunto de testes é uma série de testes repetidos.

É uma prática comum e útil criar um novo teste sempre que você encontrar um erro de programa. Você pode utilizar esse teste para verificar se a correção do erro realmente funcionou. Não descarte o teste; coloque-o na próxima versão do conjunto de testes e em todas as versões subseqüentes. Essa coleção de casos de teste é chamada de *conjunto de testes*.

O teste de regressão envolve a repetição de testes já executados para assegurar que falhas conhecidas nas versões anteriores não aparecerão em novas versões do software.

Você se surpreenderá com a freqüência com que um erro corrigido por você reaparecerá em uma versão futura. Isso é um fenômeno conhecido como *ciclo*. Às vezes você não entende muito bem a razão de um erro e aplica uma correção rápida que parece funcionar. Mais tarde, você aplica outra correção rápida que resolve um segundo problema, porém, faz o primeiro problema reaparecer. Naturalmente, sempre é melhor pensar no que realmente causa um erro e corrigir a causa na raiz do que criar uma seqüência de soluções "Band-Aid". Mas se você não for bem-sucedido ao fazer isso, pelo menos terá uma avaliação honesta de como o programa funciona. Mantendo todos os casos de teste antigos disponíveis e testando-os em cada nova versão, você obtém esse retorno. O processo de testar contra um conjunto de falhas passadas é chamado de *teste de regressão*.

Como você organiza um conjunto de testes? Uma técnica fácil é criar múltiplas classes testadoras, como `BankTester1`, `BankTester2` e assim por diante.

Uma outra abordagem útil é fornecer um testador genérico e alimentá-lo com entradas a partir de múltiplos arquivos. Considere esse testador para a classe `Bank` da Seção 7.5:

ch07/regression/BankTester.java

```
1   /**
2      Este programa testa a classe Bank.
3   */
4   public class BankTester
5   {
6      public static void main(String[] args)
7      {
8         Bank firstBankOfJava = new Bank();
9         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
10        firstBankOfJava.addAccount(new BankAccount(1015, 10000));
11        firstBankOfJava.addAccount(new BankAccount(1729, 15000));
12
13        Scanner in = new Scanner(System.in);
```

```
14
15        double threshold = in.nextDouble();
16        int c = firstBankOfJava.count(threshold);
17        System.out.println("Count: " + c);
18        int expectedCount = in.nextInt();
19        System.out.println("Expected: " + expectedCount);
20
21        int accountNumber = in.nextInt;
22        BankAccount a = firstBankOfJava.find(accountNumber);
23        if (a == null)
24           System.out.println("No matching account");
25        else
26        {
27           System.out.println("Balance of matching account: "
28                 + a.getBalance());
29           int matchingBalance = in.nextLine();
30           System.out.println("Expected: " + matchingBalance);
31        }
32     }
33  }
```

Em vez de utilizar valores fixos para o limiar e para o número de contas a serem localizadas, o programa lê esses valores e as respostas esperadas. Executando o programa com diferentes entradas, podemos testar diferentes cenários, por exemplo, diagnosticar erros "por um" (off-by-one) discutidos no Erro Comum 6.2.

Naturalmente, seria entediante digitar os valores de entrada manualmente toda vez que o teste é executado. É bem melhor salvar as entradas em um arquivo, como segue:

ch07/regression/input1.txt

```
15000
2
1015
10000
```

As interfaces de linha de comando da maioria dos sistemas operacionais fornecem uma maneira de vincular um arquivo à entrada de um programa, como se todos os caracteres no arquivo, na verdade, tivessem sido digitados por um usuário. Digite o comando a seguir em uma janela de shell:

```
java BankTester < input1.txt
```

O programa é executado, mas ele não lê mais a entrada do teclado. Em vez disso, o objeto `System.in` (e o `Scanner` que lê a partir de `System.in`) obtém a entrada do arquivo `input1.txt`. Esse processo é chamado de *redirecionamento da entrada*.

A saída continua a ser exibida na janela de console:

Saída

```
Count: 2
Expected: 2
Balance of matching account: 10000
Expected: 10000
```

Você também pode redirecionar a saída. Para salvar a saída de um programa em um arquivo, utilize o comando:

```
java BankTester < input1.txt > output1.txt
```

Isso é útil para arquivar casos de teste.

Autoverificação da aprendizagem

15. Suponha que você modificou o código para um método. Por que repetir os testes que já passaram na versão anterior do código?
16. Suponha que um cliente do seu programa localize um erro. Qual ação você deve executar além de corrigir o erro?
17. Por que o programa `BankTester` não contém *prompts* para as entradas?

Dica de produtividade 7.1

Arquivos batch e scripts de shell

A Dica de Produtividade 7.1 mostra como você pode automatizar tarefas repetitivas escrevendo arquivos batch ou scripts de shell.

Fato aleatório 7.2

Os incidentes do Therac-25

O Fato Aleatório 7.2 comenta a história do Therac-25, um aparelho computadorizado para gerar radiação no tratamento de pacientes com câncer. Devido ao mau projeto e a testes insuficientes, esse aparelho forneceu overdoses substanciais, matando alguns pacientes e mutilando seriamente outros.

RESUMO DO CAPÍTULO

1. Um array é uma seqüência de valores do mesmo tipo.
2. Você acessa elementos do array com um índice de inteiros, utilizando a notação `a[i]`.
3. Valores de índice de um array variam de 0 a `length - 1`. Acessar um elemento inexistente resulta em um erro de limite.
4. Utilize o campo `length` para encontrar o número de elementos em um array.
5. A classe `ArrayList` gerencia uma seqüência de objetos.
6. A classe `ArrayList` é uma classe genérica: `ArrayList<T>` coleciona objetos do tipo `T`.

7. Para tratar valores de tipos primitivos como objetos, você deve utilizar classes empacotadoras (*wrappers*).
8. O laço `for` aprimorado percorre todos os elementos de uma coleção.
9. Para contar os valores em uma lista de arrays, verifique todos os elementos e conte as correspondências até você alcançar o fim da lista de arrays.
10. Para localizar um valor em uma lista de arrays, verifique todos os elementos até você encontrar uma correspondência.
11. Para calcular o valor máximo ou mínimo em uma lista de arrays, inicialize um candidato com o elemento inicial. Compare então o candidato aos elementos restantes e atualize-o se encontrar um valor maior ou menor.
12. Arrays bidimensionais formam um arranjo bidimensional tabular. Você acessa elementos com um par de índices `a[i][j]`.
13. Uma variável de array armazena uma referência ao array. Copiar a variável fornece uma segunda referência ao mesmo array.
14. Utilize o método `clone` para copiar os elementos de um array.
15. Utilize o método `System.arraycopy` para copiar elementos de um array para outro.
16. Evite arrays paralelos transformando-os em arrays de objetos.
17. Um conjunto de testes é uma série de testes repetidos.
18. O teste de regressão envolve a repetição de testes já executados para assegurar que falhas conhecidas nas versões anteriores não aparecerão em novas versões do software.

LEITURA ADICIONAL

1. Barton P. Miller, Louis Fericksen, and Bryan So, "An Empirical Study of the Reliability of Unix Utilities", *Communications of the ACM*, vol. 33, no. 12 (December 1990), pp. 32–44.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.lang.Boolean
   booleanValue
java.lang.Double
   doubleValue
java.lang.Integer
   intValue
java.lang.System
   arraycopy
```

```
java.util.ArrayList<E>
   add
   get
   remove
   set
   size
```

EXERCÍCIOS DE REVISÃO

★ **Exercício R7.1.** O que é um índice? Quais são os limites de um array ou de uma lista de arrays? O que é um erro de limite?

★ **Exercício R7.2.** Escreva um programa que contenha um erro de limite. Execute o programa. O que acontece no seu computador? Como a mensagem de erro ajuda a encontrar o erro?

★★ **Exercício R7.3.** Escreva um código em Java para um laço que calcula simultaneamente os valores máximo e mínimo de uma lista de arrays. Utilize uma lista de arrays de contas bancárias como exemplo.

★ **Exercício R7.4.** Escreva um laço que lê 10 strings e as insira em uma lista de arrays. Escreva um segundo laço que imprime as strings na ordem inversa daquela em que elas foram inseridas.

★★ **Exercício R7.5.** Considere o algoritmo que utilizamos para determinar o valor máximo em uma lista de arrays. Configuramos `largestYet` como o elemento inicial, o que significou que não era mais possível utilizar o laço "*for each*". Uma abordagem alternativa é inicializar `largestYet` com `null` e então fazer um laço por todos os elementos. Naturalmente, dentro do laço você precisa testar se `largestYet` ainda é `null`. Modifique o laço que localiza a conta bancária com o maior saldo utilizando essa técnica. Essa abordagem é mais ou menos eficiente do que aquela utilizada no texto?

★★★ **Exercício R7.6.** Considere uma outra variação do algoritmo para determinar o valor máximo. Aqui, calculamos o valor máximo de um array de números.

```
double max = 0; // Contém um erro!
for (x : values)
{
   if (x > max) max = x;
}
```

Mas essa abordagem contém um erro sutil. Qual é o erro e como você pode corrigi-lo?

★ **Exercício R7.7.** Para cada um dos conjuntos de valores a seguir, escreva um código que preenche um array a com os seguintes valores.

a. 1 2 3 4 5 6 7 8 9 10

b. 0 2 4 6 8 10 12 14 16 18 20

c. 1 4 9 16 25 36 49 64 81 100

d. 0 0 0 0 0 0 0 0 0 0

e. 1 4 9 16 9 7 4 9 11

Utilize um laço quando apropriado.

★★ **Exercício R7.8.** Escreva um laço que preenche um array a com 10 números aleatórios entre 1 e 100. Escreva um código (utilizando um ou mais laços) para preencher a com 10 diferentes números aleatórios entre 1 e 100.

★ **Exercício R7.9.** O que há de errado com o laço a seguir?

```
double[] data = new double[10];
for (int i = 1; i <= 10; i++) data[i] = i * i;
```

Explique duas maneiras de corrigir o erro.

★★★T **Exercício R7.10.** Escreva um programa que constrói um array de 20 inteiros e preenche os dez primeiros elementos com os números 1, 4, 9. . . , 100. Compile-o e carregue o depurador. Depois que o array foi preenchido com três números, inspecione-o. Qual é o conteúdo dos elementos no array além daquele que você preencheu?

★★ **Exercício R7.11.** Rescreva os laços a seguir sem utilizar a construção "*for each*". Aqui, `data` é um array de valores `double`.

a.
```
for (x : data) sum = sum + x;
```

b.
```
for (x : data) if (x == target) return true;
```

c.
```
int i = 0;
for (x : data) { data [i] = 2 * x; i++; }
```

★★ **Exercício R7.12.** Reescreva os laços a seguir utilizando a construção "*for each*". Aqui, `data` é um array de valores `double`.

a.
```
for (int i = 0; i < data.length; i++) sum = sum + data[i];
```

b.
```
for (int i = 1; i < data.length; i++) sum = sum + data[i];
```

c.
```
for (int i = 0; i < data.length; i++)
    if (data[i] == target) return i;
```

★★★ **Exercício R7.13.** Dê um exemplo de:

a. Um método útil que tem um array de inteiros como um parâmetro que não é modificado.

b. Um método útil que tem um array de inteiros como um parâmetro que é modificado.

c. Um método útil que tem um array de inteiros como um valor de retorno.

Descreva cada método; não implemente os métodos.

★★★ **Exercício R7.14.** Um método que tem uma lista de arrays como um parâmetro pode alterar o conteúdo de duas maneiras. Ele pode alterar o conteúdo dos elementos individuais do array ou reorganizá-los. Descreva dois métodos úteis com parâmetros `ArrayList<BankAccount>` que alteram uma lista de arrays de objetos `BankAccount` de cada uma das duas maneiras que acabamos de descrever.

★ **Exercício R7.15.** O que são arrays paralelos? Por que arrays paralelos são indicações de uma programação ruim? Como eles podem ser evitados?

★★ **Exercício R7.16.** Como você realiza as tarefas a seguir com arrays em Java?

a. Teste se os dois arrays contêm os mesmos elementos na mesma ordem.

b. Copie um array para outro.

c. Preencha um array com zeros, sobrescrevendo todos os elementos nele.

d. Remova todos os elementos de uma lista de arrays.

★ **Exercício R7.17.** Verdadeiro ou falso?

a. Todos os elementos de um array são do mesmo tipo.
b. Subscritos de array devem ser inteiros.
c. Arrays não podem conter referências de string como elementos.
d. Arrays não podem utilizar strings como subscritos.
e. Arrays paralelos devem ter comprimento igual.
f. Arrays bidimensionais sempre têm o mesmo número de linhas e colunas.
g. Dois arrays paralelos podem ser substituídos por um array bidimensional.
h. Elementos de colunas diferentes em um array bidimensional podem ter tipos diferentes.

★★ **Exercício R7.18.** Verdadeiro ou falso?

a. Um método não pode retornar um array bidimensional.
b. Um método pode alterar o comprimento de um array que é passado como parâmetro.
c. Um método pode alterar o comprimento de uma lista de arrays que é passada como parâmetro.
d. Uma lista de arrays pode armazenar valores de qualquer tipo.

★T **Exercício R7.19.** Defina os termos *teste de regressão* e *conjunto de testes*.

★T **Exercício R7.20.** O que é o fenômeno de depuração conhecido como *ciclo*? O que você pode fazer para evitá-lo?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P7.1.** Adicione os métodos a seguir à classe `Bank`:

```
public void addAccount(int accountNumber, double initialBalance)
public void deposit(int accountNumber, double amount)
public void withdraw(int accountNumber, double amount)
public double getBalance(int accountNumber)
```

★ **Exercício P7.2.** Implemente uma classe `Purse`. Uma bolsa contém uma coleção de moedas. Por simplicidade, só armazenaremos os nomes das moedas em uma `ArrayList<String>`. (Discutiremos uma melhor representação no Capítulo 8.) Forneça um método:

```
void addCoin(String coinName)
```

Adicione um método `toString` à classe `Purse` que imprime as moedas na bolsa no formato

```
Purse[Quarter,Dime,Nickel,Dime]
```

★ **Exercício P7.3.** Escreva um método `reverse` que inverte a seqüência de moedas em uma bolsa. Utilize o método `toString` do exercício anterior para testar seu código. Por exemplo, se `reverse` for chamado com uma bolsa assim:

```
Purse[Quarter,Dime,Nickel,Dime]
```

então a bolsa muda para:

```
Purse[Dime,Nickel,Dime,Quarter]
```

★ **Exercício P7.4.** Adicione um método

```
public void transfer(Purse other)
```

que transfere o conteúdo de uma bolsa para outra. Por exemplo, se `a` for

```
Purse[Quarter,Dime,Nickel,Dime]
```

e `b` for

```
Purse[Dime,Nickel]
```

depois da chamada `a.transfer(b)`, `a` será

```
Purse[Quarter,Dime,Nickel,Dime,Dime,Nickel]
```

e `b` estará vazio.

★ **Exercício P7.5.** Escreva um método para a classe `Purse`

```
public boolean sameContents(Purse other)
```

que verifica se a outra bolsa tem as mesmas moedas na mesma ordem.

★★ **Exercício P7.6.** Escreva um método para a classe `Purse`

```
public boolean sameCoins(Purse other)
```

que verifica se a outra bolsa tem as mesmas moedas, talvez em uma ordem diferente. Por exemplo, as bolsas

```
Purse[Quarter,Dime,Nickel,Dime]
```

e

```
Purse[Nickel,Dime,Dime,Quarter]
```

devem ser consideradas iguais.

Provavelmente você precisará de um ou mais métodos auxiliares.

★★ **Exercício P7.7.** Um `Polygon` é uma curva fechada composta por segmentos de linha que unem os pontos extremos de um polígono. Implemente uma classe `Polygon` com métodos

```
public double perimeter()
```

e

```
public double area()
```

que calculem a circunferência e a área de um polígono. Para calcular o perímetro, calcule a distância entre os pontos adjacentes e some as distâncias. A área de um polígono com cantos $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ é

$$\frac{1}{2}\left(x_0 y_1 + x_1 y_2 + \cdots + x_{n-1} y_0 - y_0 x_1 - y_1 x_2 - \cdots - y_{n-1} x_0\right)$$

Como casos de teste, calcule o perímetro e a área de um retângulo e de um hexágono normal. *Nota:* Você não precisa desenhar o polígono — isso é feito no Exercício P7.15.

★ **Exercício P7.8.** Escreva um programa que lê uma seqüência de inteiros em um array e calcula a soma alternada de todos os elementos no array. Por exemplo, se o programa for executado com os dados de entrada:

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

então ele calculará:

$$1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 = -2$$

★★ **Exercício P7.9.** Escreva um programa que produz permutações aleatórias dos números 1 a 10. Para gerar uma permutação aleatória, você precisa preencher um array com os números 1 a 10 para que duas entradas do array não tenham o mesmo conteúdo. Você poderia fazer isso usando força bruta, chamando `Random.nextInt` até que ele produza um valor que ainda não está no array. Em vez disso, você deve implementar um método inteligente. Crie um segundo array e preencha-o com os números 1 a 10. Depois, selecione um desses números aleatoriamente, remova e acrescente-o ao array da permutação. Repita 10 vezes. Implemente uma classe `PermutationGenerator` com um método

```
int[] nextPermutation
```

★★ **Exercício P7.10.** Adicione um método `getWinner` à classe `TicTacToe` da Seção 7.6. Ele deve retornar `"x"` ou `"o"` para indicar um vencedor ou `" "` se ainda não houver um vencedor. Lembre-se que uma posição vitoriosa tem três marcas idênticas em uma fila, coluna ou diagonal.

★★★ **Exercício P7.11.** Escreva uma aplicação que jogue o jogo da velha. Seu programa deve desenhar o tabuleiro desse jogo, mudar de jogador a cada lance bem-sucedido e declarar o vencedor.

★★ **Exercício P7.12.** *Quadrados mágicos*. Uma matriz $n \times n$, preenchida com os números 1, 2, 3, . . . , n^2, é um quadrado mágico se a soma dos elementos em cada linha, em cada coluna e nas duas diagonais tiverem o mesmo valor. Por exemplo,

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

Escreva um programa que lê n^2 valores a partir do teclado e testa se eles formam um quadrado mágico quando organizados como uma matriz quadrada. Você precisa testar três aspectos:

- O usuário inseriu n^2 números para um dado n?
- Cada um dos números 1, 2, . . n^2 ocorre apenas uma vez na entrada de usuário?
- Quando os números são colocados em um quadrado, as somas das linhas, colunas e diagonais são idênticas entre si?

Se o tamanho da entrada for um quadrado, teste se todos os números entre 1 e n^2 estão presentes. Calcule então as somas das linhas, colunas e diagonais. Implemente uma classe `Square` com os métodos

```
public void add(int i)
public boolean isMagic()
```

★★ **Exercício P7.13.** Implemente o algoritmo a seguir para construir quadrados n-por-n^2 mágicos; isso só funciona se n for ímpar. Posicione 1 no meio da linha inferior. Depois que k foi posicionado no quadrado (i, j), adicione $k + 1$ ao quadrado à direita e abaixo, contornando as bordas. Mas se o quadrado à direita e abaixo já foi preenchido ou se você estiver no canto inferior direito, deverá então passar para o quadrado acima. Eis o quadrado 5×5 que você obtém se seguir este método:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Escreva um programa cuja entrada é o número n e cuja saída é o quadrado mágico de ordem n se n é ímpar. Implemente uma classe `MagicSquare` com um construtor que cria o quadrado e um método `toString` que retorna uma representação desse quadrado.

★G **Exercício P7.14.** Implemente uma classe `Cloud` que contém uma lista de arrays de objetos `Point2D.Double`. Métodos de suporte

```
public void add(Point2D.Double aPoint)
public void draw(Graphics2D g2)
```

Desenhe cada ponto como um pequeno círculo.

Escreva uma aplicação gráfica que desenha uma nuvem de 100 pontos aleatórios.

★★G **Exercício P7.15.** Implemente uma classe `Polygon` que contém uma lista de arrays de objetos `Point2D.Double`. Métodos de suporte

```
public void add(Point2D.Double aPoint)
public void draw(Graphics2D g2)
```

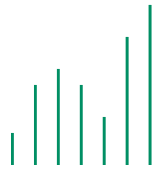
Desenhe o polígono unindo os pontos adjacentes a uma linha e então feche-o unindo os pontos inicial e final.

Escreva uma aplicação gráfica que desenha um quadrado e um pentágono utilizando dois objetos `Polygon`.

★G **Exercício P7.16.** Escreva uma classe `Chart` com métodos

```
public void add(int value)
public void draw(Graphics2D g2)
```

que exibe um gráfico de barras dos valores adicionados, como este:



Você pode assumir que os valores são posições em pixels.

★★G **Exercício P7.17.** Escreva uma classe `BarChart` com métodos

```
public void add(double value)
public void draw(Graphics2D g2)
```

que exibe um gráfico dos valores adicionados. Você pode assumir que todos os valores em `data` são positivos. Estenda as barras para que elas preencham toda a área da tela. Você precisa descobrir os valores máximos e então dimensionar cada barra.

★★★G **Exercício P7.18.** Aprimore a classe `BarChart` do Exercício P7.17 para que ela funcione corretamente quando os dados contêm valores negativos.

★★G **Exercício P7.19.** Escreva uma classe `PieChart` com métodos:

```
public void add(double value)
public void draw(Graphics2D g2)
```

que exibe um gráfico de torta dos valores em `data`. Você pode assumir que todos os valores dos dados são positivos.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 7.1.** *Simulador de pôquer*. Neste exercício, você implementará uma simulação de um popular jogo de cassino comumente chamado de vídeo pôquer. O baralho contém 52 cartas, 13 de cada naipe. No começo do jogo, as cartas são embaralhadas. Você precisa pensar em um bom método para embaralhar as cartas. (Ele não precisa ser eficiente.) As cinco cartas no topo do baralho são então apresentadas para o jogador. O jogador pode rejeitar uma, algumas ou todas as cartas. As cartas rejeitadas são substituídas por cartas provenientes do topo do baralho. Agora, o programa avalia a "mão", ou jogada. Seu programa deve declarar isso como sendo um destes:

- Nenhum par – a mão mais baixa, contendo cinco cartas de valores diferentes.
- Um par – duas cartas do mesmo valor, por exemplo duas damas.
- Dois pares – Dois pares, por exemplo, duas damas e dois 5.
- Trinca ou trio – Três cartas do mesmo valor, por exemplo, três damas.
- Seqüência (ou straight) – Cinco cartas com valores consecutivos, não necessariamente do mesmo naipe, como 4, 5, 6, 7 e 8. O ás pode vir antes de um 2 ou depois de um rei.

- Flush, naipe ou cor – Cinco cartas, não necessariamente na seqüência, do mesmo naipe.
- Full House – Uma trinca e um par, por exemplo, três damas e dois 5.
- Quadra – Quatro cartas do mesmo valor, como quatro damas.
- Straight Flush, ou seqüência de naipe e cor – Uma seqüência e um flush: Cinco cartas com valores consecutivos do mesmo naipe.
- Royal Flush – A melhor mão possível no pôquer. Um 10, um valete, uma dama, um rei e um ás, todos do mesmo naipe.

Se desejar, implemente uma aposta. O jogador paga um JavaDollar para cada jogo e ganha de acordo com a tabela de apostas a seguir:

Mão	Apostas	Mão	Apostas
Royal Flush	250	Seqüência	4
Straight Flush	50	Trinca	3
Quadra	25	Dois pares	2
Full House	6	Par de valetes ou melhor	1
Flush	5		

★★★ **Projeto 7.2.** *O Jogo da Vida* é um famoso jogo matemático que gera um comportamento incrivelmente complexo, embora ele possa ser especificado por meio de algumas regras simples. (Na verdade, ele não é um jogo no sentido tradicional, com jogadores competindo e buscando uma vitória.) Eis as regras. Joga-se em um tabuleiro retangular. Cada quadrado pode estar vazio ou ocupado. No início, você pode especificar células vazias e ocupadas como quiser; depois o jogo segue automaticamente. Em cada *geração*, a próxima geração é calculada. Uma nova célula nasce em um quadrado vazio se ela estiver cercada por exatamente três células vizinhas ocupadas. Uma célula morre de superpopulação se ela estiver cercada por quatro ou mais células vizinhas e morre de solidão se estiver cercada por zero ou uma célula vizinha. Uma célula vizinha é uma ocupante de um quadrado adjacente à esquerda, à direita, acima, abaixo ou na diagonal. A Figura 13 mostra uma célula e suas células vizinhas.

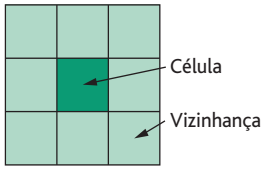


Figura 13
A vizinhança de uma célula no Jogo da Vida.

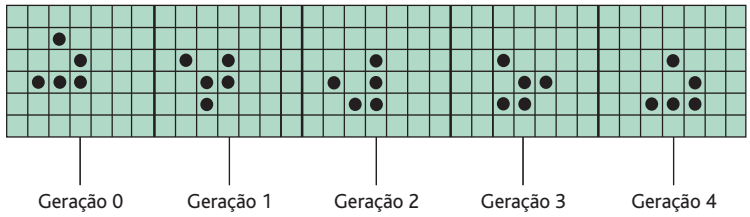


Figura 14
Planador.

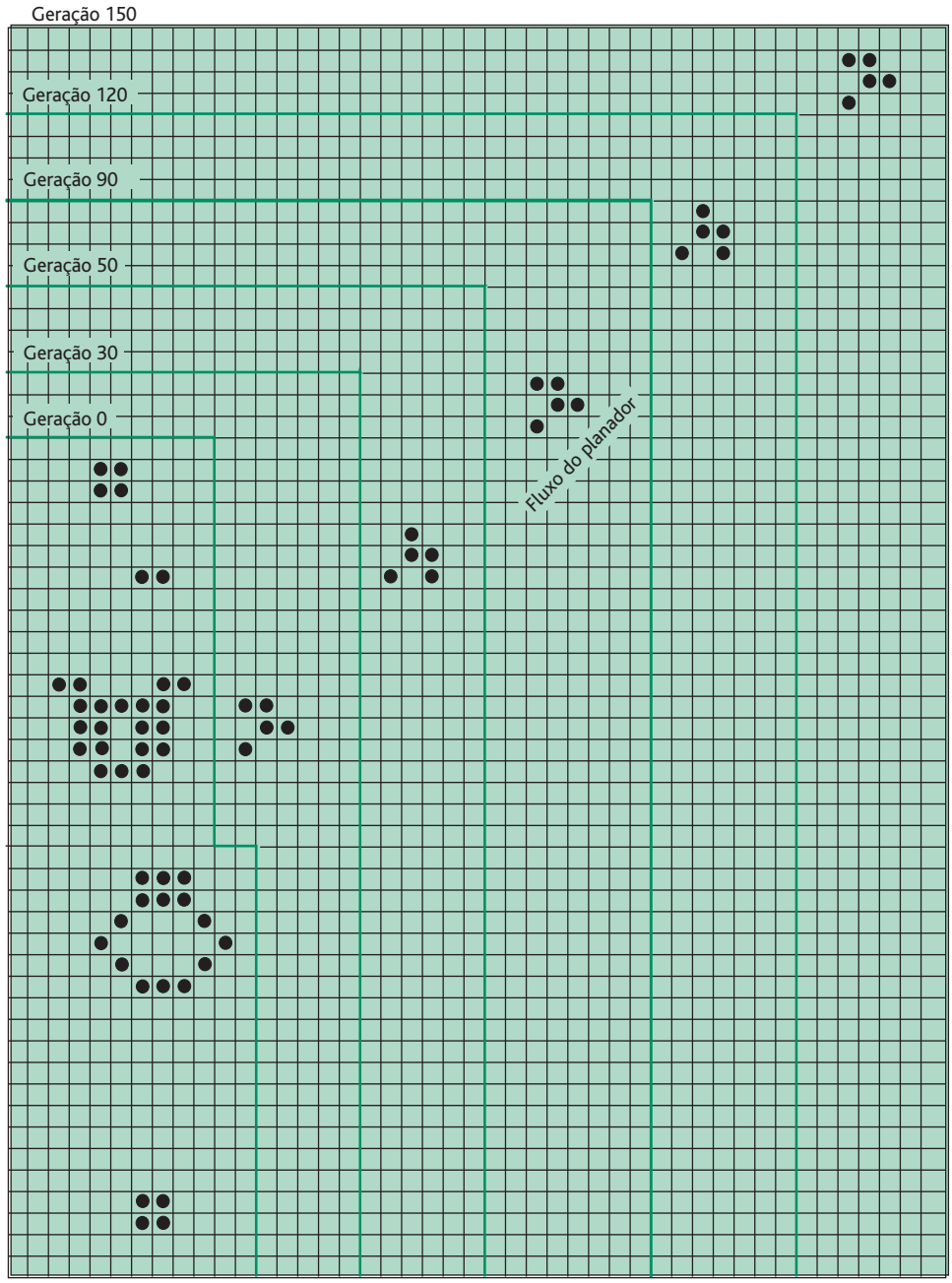


Figura 15 Arma de planador.

Muitas configurações mostram um comportamento interessante quando submetidas a essas regras. A Figura 14 mostra um *planador* (uma das várias formas identificadas pela disposição das células), observado ao longo de cinco gerações. Observe como ele se move. Depois de quatro gerações, ele é transformado em uma forma idêntica, mas localizado a um quadrado à direita e abaixo.

Um das configurações mais surpreendentes é o glider gun (arma de "planador"): uma coleção complexa de células que, depois de 30 movimentos, se autotransforma nela mesma e num planador (veja Figura 15).

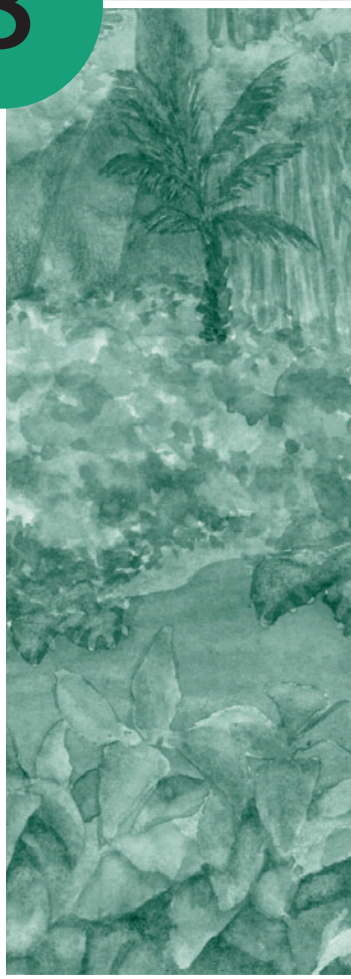
Programe esse jogo para eliminar o trabalho de calcular gerações sucessivas manualmente. Utilize um array bidimensional para armazenar a configuração retangular. Escreva um programa que mostra gerações sucessivas do jogo. Você pode ganhar crédito extra se implementar uma aplicação gráfica que permite que o usuário adicione ou remova células clicando com o mouse.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, mas *não* 100.
2. a. 0
   b. um erro em tempo de execução: índice de array fora dos limites
   c. um erro em tempo de compilação: `c` não é inicializado
3. `new String[10];`
   `new ArrayList<String>();`
4. `names` contém as strings `"B"` e `"C"` nas posições 0 e 1.
5. `double` é um dos oito tipos primitivos. `Double` é um tipo de classe.
6. `data.set(0, data.get(0) + 1);`
7. `for (double x : data) System.out.println(x);`
8. O laço escreve um valor em `data[i]`. O laço "*for each*" não tem a variável de índice `i`.
9. Ele retorna a primeira correspondência que localiza.
10. Sim, mas a primeira comparação sempre falharia.
11. `int[][] array = new int[4][4];`
12.
```
int count = 0;
for (int i = 0; i < ROWS; i++)
   for (int j = 0; j < COLUMNS; j++)
      if (board[i][j].equals(" ")) count++;
```
13. Utilize os métodos `add` e `remove`.
14. Alocar um novo array e copiar os elementos é demorado. Você não iria querer passar pelo processo toda vez que adiciona um elemento.
15. É possível introduzir erros ao modificar o código.
16. Adicione um caso de teste ao conjunto de testes que verifica se o erro foi corrigido.
17. Nenhum usuário humano veria os prompts porque a entrada é fornecida a partir de um arquivo.

Capítulo 8

Projetando Classes



Objetivos do capítulo

- Aprender a escolher classes apropriadas para implementar
- Entender os conceitos de coesão e acoplamento
- Minimizar o uso de efeitos colaterais
- Documentar as responsabilidades dos métodos e seus chamadores com pré-condições e pós-condições
- Entender a diferença entre métodos de instância e métodos estáticos
- Introduzir o conceito de campos estáticos
- Entender as regras de escopo para variáveis locais e para campos de instância
- Aprender sobre pacotes

T Aprender sobre frameworks de teste de unidade

Neste capítulo, você aprenderá mais sobre como projetar classes. Primeiro, discutiremos o processo de descobrir classes e definir métodos. Em seguida, examinaremos como os conceitos de pré e pós-condições permitem especificar, implementar e invocar métodos corretamente. Você também aprenderá várias questões mais técnicas, como métodos estáticos e variáveis. Por fim, veremos como utilizar pacotes para organizar suas classes.

Conteúdo do Capítulo

8.1 Escolhendo classes 316

8.2 Coesão e acoplamento 318
Dica de qualidade 8.1: Consistência 320

8.3 Classes modificadoras, classes de acesso e classes imutáveis 321

8.4 Efeitos colaterais 321
Erro comum 8.1: Tentando modificar parâmetros de tipos primitivos 323
Dica de qualidade 8.2: Minimize os efeitos colaterais 325
Dica de qualidade 8.3: Não altere o conteúdo das variáveis de parâmetros 325
Tópico avançado 8.1: Chamada por valor e chamada por referência ⊕

8.5 Pré-condições e pós-condições 326
Sintaxe 8.1: Assertiva 327
Tópico avançado 8.2: Invariantes de classe ⊕

8.6 Métodos estáticos 329

8.7 Campos estáticos 331
Tópico avançado 8.3: Formas alternativas de inicialização de campos ⊕

8.8 Escopo 334
Erro comum 8.2: Sombreamento 337
Dica de produtividade 8.1: Pesquisa e substituição global ⊕
Dica de produtividade 8.2: Expressões regulares ⊕
Tópico avançado 8.4: Importações estáticas ⊕

8.9 Pacotes 339
Sintaxe 8.2: Especificação de pacote 340
Erro comum 8.3: Uso confuso de pontos 343
Como fazer 8.1: Programando com pacotes 346
Fato aleatório 8.1: O crescimento explosivo dos computadores pessoais ⊕

8.10T Frameworks de teste de unidade 345

8.1 Escolhendo classes

Você já utilizou um bom número de classes nos capítulos anteriores e provavelmente projetou algumas como parte de seus exercícios de programação. Projetar uma classe pode ser um desafio – nem sempre é fácil dizer como começar ou se o resultado é de boa qualidade.

Estudantes com alguma experiência anterior em programação, em uma outra linguagem, estão acostumados a programar *funções*. Uma função executa uma ação. Na programação orientada a objetos, as ações aparecem como métodos. Cada método, porém, pertence a uma classe. Classes são coleções de objetos e objetos não são ações – eles são entidades. Portanto, você precisa começar a atividade de programação identificando objetos e as classes às quais eles pertencem.

Lembre-se da regra geral do Capítulo 2: nomes de classe devem ser substantivos e nomes de método devem ser verbos.

Uma classe deve representar um único conceito do domínio do problema, como negócios, ciência ou matemática.

O que faz uma boa classe? Acima de tudo, uma classe deve *representar um único conceito.* Algumas classes que vimos representam conceitos matemáticos:

- `Point`
- `Rectangle`
- `Ellipse`

Outras classes são abstrações de situações da vida real.

- `BankAccount`
- `CashRegister`

Para essas classes, as propriedades de um objeto típico são fáceis de entender. Um objeto `Rectangle` tem uma largura e uma altura. Dado um objeto `BankAccount`, você pode depositar e sacar dinheiro. Geralmente, conceitos da parte do universo compreendida pelo programa, como ciências, negócios ou um jogo, fazem boas classes. O nome para essa classe deve ser um substantivo que descreve o conceito. Alguns nomes de classes padrão em Java são um pouco estranhos, como `Ellipse2D.Double`, mas você pode escolher nomes mais apropriados para suas classes.

Uma outra categoria útil de classe pode ser descrita como *atores*. Objetos de uma classe de atores realizam alguns trabalhos para você. Exemplos de atores são a classe `Scanner` do Capítulo 4 e a classe `Random` no Capítulo 6. Um objeto `Scanner` varre um fluxo de dados em busca de números e strings. Um objeto `Random` gera números aleatórios. Se utilizar a língua inglesa para atribuir nomes, uma boa idéia é escolher nomes de classes para os atores que terminem em "er" ou "or". (Um nome melhor para a classe `Random` poderia ser `RandomNumberGenerator`.)

Ocasionalmente, uma classe não tem nenhum objeto, mas contém uma coleção de constantes e métodos estáticos relacionados. A classe `Math` é um exemplo típico. Uma classe assim é chamada de *classe utilitária*.

Por fim, você viu classes com somente um método `main`. O único propósito dessas classes é iniciar um programa. Da perspectiva de projeto, estes são exemplos mais ou menos degenerados de classes.

O que não poderia ser uma boa classe? Se você não pode afirmar a partir do nome da classe o que um objeto da classe supostamente deve fazer, provavelmente você não está no caminho certo. Por exemplo, sua lição de casa poderia solicitar para você escrever um programa que imprima cheques de pagamento. Suponha que inicialmente você tente projetar uma classe `PaycheckProgram`. O que um objeto dessa classe faria? Um objeto dessa classe teria que fazer tudo o que a lição de casa exige fazer. Isso não simplifica nada. Uma classe melhor seria `Paycheck`. Seu programa pode então manipular um ou mais objetos da classe `Paycheck`.

Um outro equívoco comum, especialmente de estudantes habituados a escrever programas que consistem em funções, é transformar uma ação em uma classe. Por exemplo, se sua lição de casa fosse calcular um cheque de pagamento, você poderia considerar escrever uma classe `ComputePaycheck`. Mas você consegue visualizar um objeto do tipo ComputePaycheck? O fato de que ComputePaycheck não é um substantivo leva você a achar que está no caminho errado. Por outro lado, uma classe `Paycheck` tem um sentido intuitivo. O termo "cheque de pagamento" (ou Paycheck) é um substantivo. Você pode visualizar um objeto de cheque de pagamento. Então você pode pensar em métodos úteis para a classe `Paycheck`, como `computeTaxes`, que o ajudarão a resolver a lição de casa.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Qual é a regra geral para escolher classes?
2. Seu trabalho é escrever um programa de xadrez. A classe ChessBoard (`Tabuleiro`) poderia ser uma classe apropriada? Que tal `MovePiece`?

8.2 Coesão e acoplamento

Nesta seção você aprenderá dois critérios úteis para analisar a qualidade da interface pública de uma classe.

A interface pública de uma classe é coesa se todos os seus recursos estiverem relacionados ao conceito que a classe representa.

Uma classe deve representar um único conceito. As constantes e os métodos públicos que a interface pública expõe devem ser *coesos*. Isto é, todos os recursos da interface devem estar intimamente relacionados ao único conceito que a classe representa.

Se achar que a interface pública de uma classe referencia vários conceitos, isso é um bom sinal de que pode ser a hora de utilizar classes separadas. Considere, por exemplo, a interface pública da classe `CashRegister` no Capítulo 4:

```
public class CashRegister
{
   public void enterPayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   . . .
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   . . .
}
```

Na verdade há dois conceitos aqui: uma caixa registradora que armazena moedas e calcula o total e os valores das moedas individuais. (Por simplicidade, supomos que a caixa registradora só contém moedas, não cédulas. O Exercício P8.1 discute uma solução mais geral.)

Faz sentido ter uma classe `Coin` separada e tornar as moedas responsáveis pelos seus valores.

```
public class Coin
{
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
```

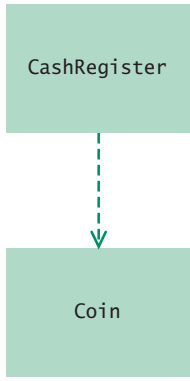


Figura 1
Relacionamento de dependência entre as classes `CashRegister` e `Coin`.

A classe `CashRegister` pode então ser simplificada:

```
public class CashRegister
{
   public void enterPayment(int coinCount, Coin coinType) { . . . }
   . . .
}
```

Agora a classe `CashRegister` não precisa conhecer mais nada sobre os valores das moedas. A mesma classe pode muito bem tratar euros ou zorkmids!

Claramente essa é uma solução melhor, porque ela separa as responsabilidades entre a caixa registradora e as moedas. A única razão por que não seguimos essa abordagem no Capítulo 4 foi manter o exemplo da `CashRegister` simples.

Uma classe depende de outra classe se ela usa objetos dessa classe.

Muitas classes precisam de outras classes para que possam fazer o seu trabalho. Por exemplo, a classe `CashRegister` reestruturada agora depende da classe `Coin` para determinar o valor do pagamento.

Para visualizar os relacionamentos, como a dependência entre as classes, programadores criam diagramas de classes. Neste livro, utilizamos a notação UML ("Unified Modeling Language") para objetos e classes. UML é uma notação para análise orientada a objetos cujo projeto foi criado por Grady Booch, Ivar Jacobson e James Rumbaugh, três importantes pesquisadores em desenvolvimento de software orientado a objetos. A notação UML distingue entre *diagramas de objetos* e diagramas de classes. Em um diagrama de objetos os nomes das classes são sublinhados; em um diagrama de classes os nomes das classes não são sublinhados. Em um diagrama de classes, você indica a dependência por meio de uma linha tracejada usando uma seta com a ponta aberta apontando para a classe dependente. A Figura 1 mostra um diagrama de classes que indica que a classe `CashRegister` depende da classe `Coin`.

Observe que a classe `Coin` *não* depende da classe `CashRegister`. Moedas não fazem idéia de que são coletadas pela caixa registradora e elas podem realizar suas funções sem nunca chamar um método qualquer na classe `CashRegister`.

Se muitas classes de um programa dependerem umas das outras, dizemos então que o *acoplamento* entre as classes é alto. Inversamente, se houver poucas dependências entre as classes, dizemos que o acoplamento é baixo (veja Figura 2).

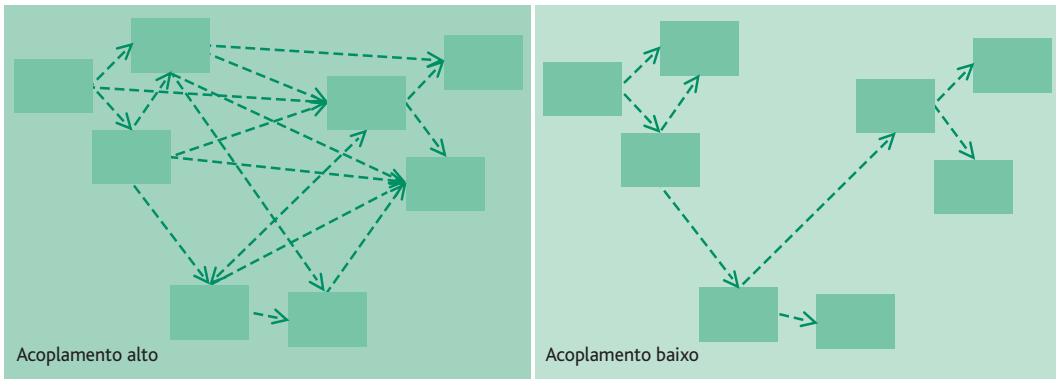


Figura 2 Acoplamento alto e acoplamento baixo entre classes.

Uma boa prática é minimizar o acoplamento (isto é, a dependência) entre classes.

Por que o acoplamento é importante? Se a classe `Coin` muda na próxima distribuição do programa, todas as classes que dependem dela podem ser afetadas. Se a alteração for drástica, todas as classes associadas devem ser atualizadas. Além disso, se quisermos utilizar uma classe em outro programa, teremos que levar junto todas as classes das quais ela depende. Portanto, queremos remover o acoplamento desnecessário entre as classes.

Autoverificação da aprendizagem

3. Por que a classe `CashRegister` do Capítulo 4 não é coesa?
4. Por que a classe `Coin` não depende da classe `CashRegister`?
5. Por que o acoplamento deve ser minimizado entre as classes?

Dica de qualidade 8.1

Consistência

Nesta seção você aprendeu dois critérios para analisar a qualidade da interface pública de uma classe. Você deve maximizar a coesão e eliminar o acoplamento desnecessário. Há um outro critério que gostaríamos que você prestasse atenção – *consistência*. Quando você tem vários métodos, siga um esquema consistente para os nomes e parâmetros desses métodos. Isso é simplesmente um sinal de bom estilo.

Infelizmente, você pode localizar várias inconsistências na biblioteca padrão. Eis um exemplo. Para mostrar uma caixa de diálogo de entrada, você chama

```
JOptionPane.showInputDialog(promptString)
```

Para mostrar uma caixa de diálogo de mensagem, você chama

```
JOptionPane.showMessageDialog(null, messageString)
```

O que é o parâmetro `null`? Sabemos que o método `showMessageDialog` precisa de um parâmetro para especificar a janela pai ou `null` se nenhuma janela pai for requerida. Mas o método `showInputDialog` não requer nenhuma janela pai. Por que essa inconsistência? Não há razão alguma. Seria mais fácil fornecer um método `showMessageDialog` que espelhasse exatamente o método `showInputDialog`.

Inconsistências como essas não são uma falha fatal, mas são uma dor de cabeça, especialmente porque podem ser facilmente evitadas.

8.3 Classes modificadoras, classes de acesso e classes imutáveis

Lembre-se de que um *método modificador* altera o objeto a partir do qual ele é invocado, enquanto um *método de acesso* meramente acessa informações sem fazer nenhuma modificação. Por exemplo, na classe `BankAccount`, os métodos `deposit` e `withdraw` são métodos modificadores. Chamar

```
account.deposit(1000);
```

modifica o estado do objeto `account`, mas a chamada

```
double balance = account.getBalance();
```

não modifica o estado de `account`.

Uma classe imutável não tem nenhum método modificador.

Você pode chamar um método de acesso quantas vezes quiser – você sempre obtém a mesma resposta e ele não altera o estado do seu objeto. Isso é claramente uma propriedade desejável, porque torna o comportamento desse método bem previsível. Algumas classes foram projetadas para terem apenas métodos de acesso e absolutamente nenhum método modificador. Essas classes são chamadas *imutáveis*. Um exemplo é a classe `String`. Depois de uma string ser construída, seu conteúdo nunca muda. Nenhum método na classe `String` pode modificar o conteúdo de uma string. Por exemplo, o método `toUpperCase` não altera os caracteres na string original. Em vez disso, ele constrói uma *nova* string que contém os caracteres em letras maiúsculas:

```
String name = "John Q. Public";
String uppercased = name.toUpperCase(); // name não é alterado
```

Uma classe imutável tem uma vantagem importante: é seguro fornecer referências aos seus objetos livremente. Se nenhum método puder alterar o valor do objeto, nenhum código poderá modificar o objeto em um momento inesperado. Em comparação, se fornecer uma referência `BankAccount` a um outro método qualquer, você terá de estar ciente de que o estado do seu objeto pode mudar – o outro método pode chamar os métodos `deposit` e `withdraw` na referência que você forneceu.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

6. O método `substring` da classe `String` é um método de acesso ou modificador?
7. A classe `Rectangle` é imutável?

8.4 Efeitos colaterais

Um método modificador altera o objeto a partir do qual ele é invocado e um método de acesso deixa-o inalterado. Essa classificação só é válida para o objeto a partir do qual o método é invocado.

Um efeito colateral de um método é qualquer modificação nos dados que possa ser observada externamente.

Um *efeito colateral* de um método é qualquer tipo de modificação nos dados que possa ser observada fora do método. Os métodos modificadores têm um efeito colateral, a saber, a modificação do parâmetro implícito.

Eis um exemplo de um método com outro tipo de efeito colateral, a atualização de um parâmetro explícito:

```
public class BankAccount
{
   /**
      Transfere dinheiro dessa conta para uma outra conta.
      @param amount quantia de dinheiro a transferir
      @param other conta da qual o dinheiro será transferido
   */
   public void transfer(double amount, BankAccount other)
   {
      balance = balance - amount;
      other.balance = other.balance + amount;
   }
   . . .
}
```

Você deve minimizar os efeitos colaterais que vão além da modificação do parâmetro implícito.

Como regra geral, atualizar um parâmetro explícito pode ser um fato surpreendente para os programadores, e é melhor evitar isso sempre que possível.

Um outro exemplo de um efeito colateral é a saída. Considere a forma como sempre imprimimos um saldo bancário:

```
System.out.println("The balance is now $"
      + momsSavings.getBalance());
```

Por que simplesmente não temos um método `printBalance`?

```
public void printBalance() // Não recomendável
{
   System.out.println("The balance is now $" + balance);
}
```

Isso seria mais conveniente se você realmente quisesse imprimir o valor. Mas, naturalmente, há casos em que você quer o valor para algum outro propósito. Portanto, você não pode simplesmente descartar o método `getBalance` em troca de `printBalance`.

Mais importante, o método `printBalance` força suposições fortes na classe `BankAccount`.

- A mensagem está em inglês – você supõe que o usuário do seu software leia em inglês. A maioria das pessoas no planeta não lê.
- Você usa `System.out`. Um método que use `System.out` não funcionará em um sistema embarcado, como o computador dentro de um caixa automático.

Em outras palavras, esse projeto viola a regra da minimização do acoplamento das classes. O método `printBalance` associa a classe `BankAccount` com as classes `System` e `PrintStream`. É melhor desacoplar a entrada/saída do funcionamento real das suas classes.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

8. Se a referenciar uma conta bancária, então a chamada `a.deposit(100)` modifica o objeto conta bancária. Isso é um efeito colateral?
9. Considere a classe `DataSet` do Capítulo 6. Suponha que fôssemos adicionar um método:

```
void read(Scanner in)
{
   while (in.hasNextDouble())
      add(in.nextDouble());
}
```

Esse método tem um efeito colateral além de modificar o conjunto de dados?

ERRO COMUM 8.1

Tentando modificar parâmetros de tipos primitivos

Métodos não podem atualizar parâmetros de tipos primitivos (números, `char` e `boolean`). Para ilustrar esse ponto, vamos tentar escrever um método que atualiza um parâmetro numérico:

```
public class BankAccount
{
   /**
      Transfere dinheiro a partir dessa conta e tenta adicioná-lo a um saldo.
      @param amount quantia de dinheiro a transferir
      @param otherBalance saldo ao qual adicionar a quantia
   */
void transfer(double amount, double otherBalance) 2
    {
      balance = balance - amount;
      otherBalance = otherBalance + amount;
         // Não funcionará
    } 3
    . . .
}
```

Isso não funciona. Vamos considerar uma chamada de método.

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); 1
System.out.println(savingsBalance); 4
```

> Em Java, um método nunca pode alterar parâmetros de tipos primitivos.

Quando o método inicia, a variável de parâmetro `otherBalance` é configurada com o mesmo valor de `savingsBalance`. Então o valor de `otherBalance` é modificado, mas essa modificação não tem nenhum efeito sobre `savingsBalance`, porque `otherBalance` é uma variável separada (veja Figura 3). Quando o método termina, a variável `otherBalance` morre e `savingsBalance` não é incrementado.

Por que o exemplo no começo da Seção 8.4 funcionou, no qual o segundo parâmetro explícito era uma referência a `BankAccount`? A variável de parâmetro continha uma cópia da referência ao objeto. Por meio dessa referência, o método é capaz de modificar o objeto.

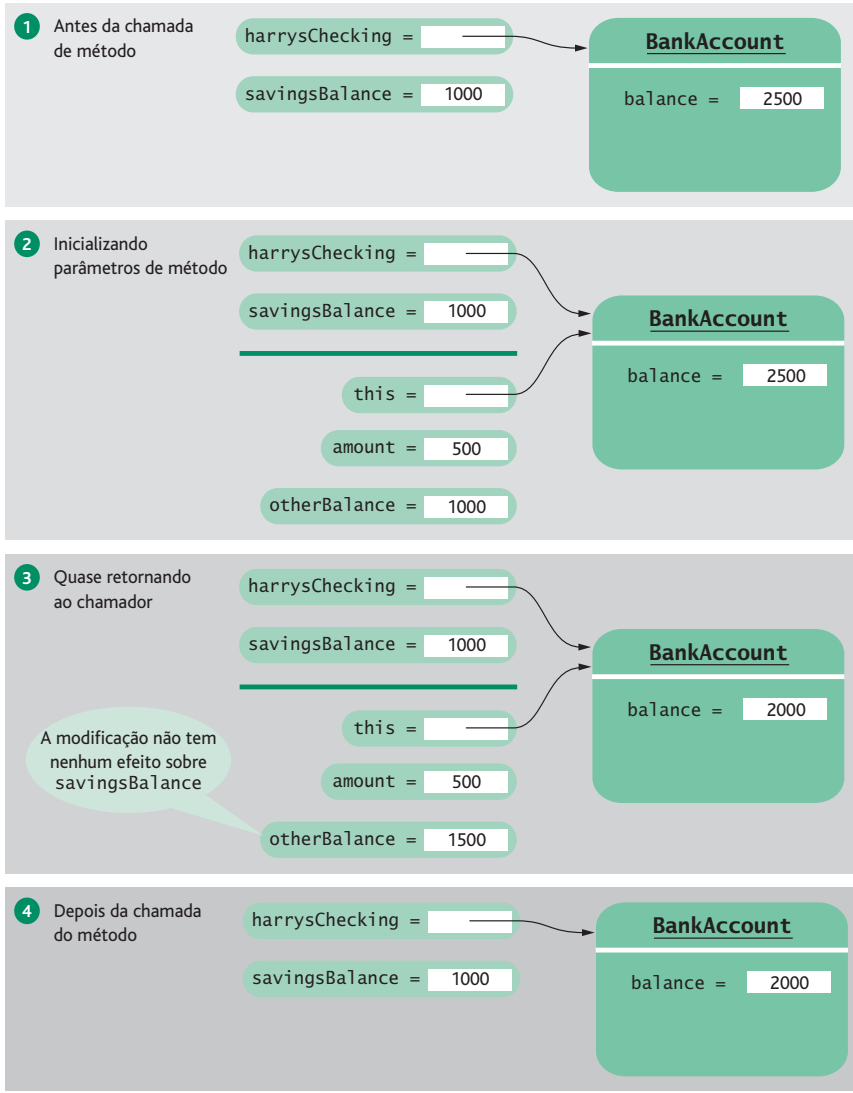


Figura 3 Modificar um parâmetro numérico não tem nenhum efeito sobre o chamador.

Já vimos essa diferença entre objetos e tipos primitivos no Capítulo 2. Como conseqüência, um método Java *nunca* pode modificar os números que ele recebe.

DICA DE QUALIDADE 8.2

Minimize os efeitos colaterais

Em um mundo ideal, todos os métodos seriam de acesso, simplesmente retornariam uma resposta sem alterar absolutamente nenhum valor. (De fato, programas escritos nas chamadas linguagens de programação *funcional*, como Scheme e ML, chegam perto desse mundo ideal.) Naturalmente, em uma linguagem de programação orientada a objetos, utilizamos objetos para lembrar das alterações de estado. Portanto, um método que apenas altera o estado do seu parâmetro implícito certamente é aceitável. Embora efeitos colaterais não possam ser completamente eliminados, eles podem ser a fonte de surpresas e problemas e devem ser minimizados. Eis uma classificação do comportamento de métodos.

- Métodos de acesso sem nenhuma alteração em qualquer parâmetro explícito – nenhum efeito colateral. Exemplo: `getBalance`.
- Métodos modificadores sem nenhuma alteração em qualquer parâmetro explícito – um efeito colateral aceitável. Exemplo: `withdraw`.
- Métodos que alteram um parâmetro explícito – um efeito colateral que deve ser evitado quando possível. Exemplo: `transfer`.
- Métodos que alteram outro objeto (como `System.out`) – um efeito colateral que deve ser evitado. Exemplo: `printBalance`.

DICA DE QUALIDADE 8.3

Não altere o conteúdo das variáveis de parâmetro

Como explicado no Erro comum 8.1 e no Tópico avançado 8.1, um método pode tratar suas variáveis de parâmetro como qualquer outra variável local e alterar seu conteúdo. Entretanto, essa alteração só afeta a variável de parâmetro dentro do próprio método – não altera nenhum valor fornecido na chamada do método. Alguns programadores tiram proveito da natureza temporária das variáveis de parâmetro utilizando-as como armazenadores "convenientes" para resultados intermediários, como neste exemplo:

```
public void deposit(double amount)
{
   // Utilizando a variável de parâmetro para armazenar um valor intermediário
   amount = balance + amount;    // Estilo ruim
   . . .
}
```

Esse código produziria erros se uma outra instrução no método referenciasse `amount` esperando que ele fosse o valor do parâmetro. Ele também confundiria outros programadores ao manter esse método. Você sempre deve tratar as variáveis de parâmetro como se fossem constantes. Não atribua novos valores a elas. Em vez disso, introduza uma nova variável local.

```
public void deposit(double amount)
{
   double newBalance = balance + amount;
   . . .
}
```

TÓPICO AVANÇADO 8.1

Chamada por valor e chamada por referência

O Tópico Avançado 8.1 explica os conceitos teóricos da "chamada por valor" e da "chamada por referência" e demonstra que a linguagem de programação Java sempre usa a "chamada por valor".

8.5 Pré-condições e pós-condições

Uma pré-condição é um requisito que o chamador de um método deve cumprir. Se um método for chamado violando uma pré-condição, esse método não será responsável por calcular o resultado correto.

Uma *pré-condição* é um requisito a que o chamador de um método deve obedecer. Por exemplo, o método `deposit` da classe `BankAccount` tem uma pré-condição de que a quantia a ser depositada não deve ser negativa. É responsabilidade do chamador nunca chamar um método se uma das suas pré-condições for violada. Se, mesmo assim, o método for chamado, ele não será responsável por produzir um resultado correto.

Portanto, uma pré-condição é uma parte importante do método e você deve documentá-la. Aqui, documentamos a pré-condição de que o parâmetro `amount` não deve ser negativo.

```
/**
   Deposita dinheiro nessa conta.
   @param amount quantia de dinheiro a depositar
   (Pré-condição: amount >= 0)
*/
```

Algumas extensões `javadoc` suportam uma marca `@precondition` ou `@requires`, mas ela não é parte do programa `javadoc` padrão. Como a ferramenta `javadoc` padrão pula todas as marcas desconhecidas, simplesmente adicionamos a pré-condição à explicação do método ou à marca `@param` apropriada.

Pré-condições geralmente são utilizadas por uma de duas razões:

1. Restringir os parâmetros de um método.
2. Exigir que um método somente seja chamado quando estiver no *estado* apropriado.

Por exemplo, depois de um `Scanner` ter verificado todas as entradas, não será mais válido chamar o método `next`. Portanto, uma pré-condição para o método `next` é que o método `hasNext` retorne `true`.

Um método é responsável por operar corretamente somente quando seu chamador atendeu a todas as pré-condições. O método é livre para fazer *qualquer coisa* se uma pré-condição não for atendida. Seria perfeitamente válido o método reformatar o disco rígido sempre que fosse chamado com uma entrada errada. Naturalmente, isso não é sensato. O que um método realmente deve fazer quando é chamado com entradas impróprias? Por exemplo, o que `account.deposit(-1000)` deve fazer? Há duas escolhas.

1. Um método pode verificar a violação e *lançar uma exceção*. O método então não retorna ao seu chamador; em vez disso, o controle é transferido para uma rotina de tratamento de exceção. Se não houver nenhuma rotina de tratamento de exceção, o programa terminará. Discutiremos exceções no Capítulo 11.
2. Um método pode pular a verificação e funcionar sob a suposição de que as pré-condições foram atendidas. Se elas não forem atendidas, qualquer dado impróprio (como um saldo negativo) ou outras falhas serão responsabilidade do chamador.

A primeira abordagem pode ser ineficiente, particularmente se a mesma verificação for executada muitas vezes por vários métodos. A segunda abordagem pode ser perigosa. O mecanismo de *assertiva* foi criado para fornecer o melhor das duas abordagens.

Uma assertiva é uma condição lógica em um programa que você acredita ser verdade.

Uma assertiva é uma condição que você acredita ser sempre verdade em uma determinada localização do programa. Uma verificação de assertiva testa se uma assertiva é verdadeira. Eis uma verificação de assertiva típica que testa uma pré-condição:

```
public double deposit (double amount)
{
   assert amount >= 0;
   balance = balance + amount;
}
```

Nesse método, o programador espera que a quantia `amount` nunca seja negativa. Quando a assertiva está correta, nada de errado acontece e o programa funciona da maneira normal. Se, por alguma razão, a assertiva falhar e a *verificação de assertiva estiver ativada*, o programa terminará com um `AssertionError`.

Mas se a verificação de assertiva estiver desativada, a assertiva nunca será verificada e o programa executará à velocidade máxima. Por padrão, a verificação de assertiva permanece desativada quando você executa um programa. Para executar um programa com a verificação de assertiva ativada, use este comando:

```
java -enableassertions MyProg
```

Você também pode utilizar o atalho `-ea` em vez de `-enableassertions`. Definitivamente, é recomendável ativar a verificação de assertiva durante o desenvolvimento e teste de programas.

SINTAXE 8.1 Assertiva

`assert` *condição*`;`

Exemplo:

```
assert amount >= 0;
```

Objetivo:

Afirmar que uma condição foi atendida. Se a verificação de assertiva estiver ativada e a condição for falsa, é lançado um erro de assertiva.

Você não precisa utilizar assertivas para verificar pré-condições – lançar uma exceção é uma outra opção razoável. Mas assertivas têm uma vantagem: você pode desativá-las depois de testar seu programa para que ele execute à velocidade máxima. Dessa maneira, você nunca se sentirá contrariado por adicionar uma grande quantidade de assertivas ao seu código. Você também pode utilizar assertivas para verificar condições além das pré-condições.

Muitos programadores iniciantes acham que não é "elegante" abortar o programa quando uma pré-condição é violada. Por que, em vez disso, simplesmente não retornar ao chamador?

```
public void deposit(double amount)
{
   if (amount < 0)
      return; // Não recomendável
   balance = balance + amount;
}
```

Isso é válido – afinal de contas, um método pode fazer qualquer coisa se suas pré-condições forem violadas. Mas não é tão bom quanto uma verificação de assertiva. Se o programa que chama o método `deposit` tiver erros que fazem com que ele passe uma quantia negativa como um valor de entrada, a versão que gera uma falha de assertiva tornará os erros bem óbvios durante o teste – é difícil ignorá-los quando o programa aborta. A versão silenciosa, por outro lado, não emitirá um alerta e, como conseqüência, talvez você não perceba que ele realiza alguns cálculos errados. Pense nas assertivas como um tipo de "remédio amargo" para a verificação de pré-condições.

Se um método foi chamado de acordo com suas pré-condições, ele então precisa assegurar que suas pós-condições serão válidas.

Quando um método é chamado em conformidade com suas pré-condições, então esse método promete fazer o trabalho corretamente. Um tipo diferente de promessa que o método faz é chamado pós-condição. Há dois tipos de pós-condições:

1. O valor de retorno é calculado corretamente.
2. O objeto permanece em um determinado estado depois que a chamada ao método é completada.

Eis uma pós-condição que faz uma afirmação sobre o estado do objeto depois de o método `deposit` ser chamado.

```
/**
   Deposita dinheiro nessa conta.
   (Pós-condição: getBalance() >= 0)
   @param amount quantia de dinheiro a depositar
   (Pré-condição: amount >= 0)
*/
```

Contanto que a pré-condição seja atendida, esse método garante que o saldo depois do depósito não seja negativo.

Algumas extensões do `javadoc` suportam uma marca `@postcondition` ou `@ensures`. Mas, assim como ocorre com as pré-condições, simplesmente adicionamos pós-condições à explicação do método ou à marca `@return`, porque o programa `javadoc` padrão pula todas as marcas que ele não conhece.

Alguns programadores acham necessário especificar uma pós-condição para cada método. Mas quando você usa `javadoc`, uma parte da pós-condição na marca `@return` já é especificada e você não deve repeti-la em uma pós-condição.

```
// Essa instrução de pós-condição é redundante.
/**
   Retorna o saldo atual dessa conta.
   @return saldo da conta
   (Pós-condição: O valor de retorno é igual ao saldo da conta.)
*/
```

Observe que só formulamos pré e pós-condições em termos da *interface* da classe. Portanto, declaramos a pré-condição do método `withdraw` como `amount <= getBalance()`, não `amount <= balance`. Afinal de contas, o chamador, que precisa verificar a pré-condição, só tem acesso à interface pública, não à implementação privada.

Bertrand Meyer [1] compara pré-condições e pós-condições a contratos. Na vida real, contratos descrevem as obrigações das partes contratantes. Por exemplo, um mecânico poderia prometer concertar os freios do seu carro e você, por sua vez, prometeria pagar certa quantia em dinheiro. Se uma das partes quebrar a promessa, então a outra não estará presa aos termos do contrato. Da mesma maneira, pré e pós-condições são termos contratuais entre um método e seu chamador. O método promete atender a pós-condição para todas as entradas que atendam a pré-condição. O chamador promete nunca chamar o método com entradas inválidas. Se o chamador cumprir sua promessa e obtiver uma resposta errada, ele poderá levar o método ao "tribunal do programador". Se o chamador não cumprir sua promessa e, como conseqüência, algo terrível acontecer, ele não terá nenhum recurso.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

10. Por que você poderia querer adicionar uma pré-condição a um método que você fornece a outros programadores?
11. Quando você implementa um método com uma pré-condição e percebe que o chamador não cumpriu a pré-condição, você precisa notificar o chamador?

TÓPICO AVANÇADO 8.2

Invariantes de classe

O Tópico Avançado 8.2 introduz o tópico das invariantes de classe, instruções lógicas que são verdadeiras após cada chamada de método ou construtor. Invariantes de classe podem ser utilizadas para provas de correção.

8.6 Métodos estáticos

Um método estático não é invocado em um objeto.

Às vezes você precisa de um método que não seja invocado a partir de um objeto. Chamamos esse método de método estático ou de *método de classe*. Por outro lado, os métodos que você escreveu até agora são freqüentemente chamados de *métodos de instância* porque operam em uma instância particular de um objeto.

Um exemplo típico de um método estático é o método `sqrt` da classe `Math`. Quando você chama `Math.sqrt(x)`, você não fornece nenhum parâmetro implícito. (Lembre-se de que `Math` é o nome de uma classe, não de um objeto.)

Por que você iria querer escrever um método que não opera em um objeto? A razão mais comum é encapsular algum cálculo que envolve apenas números. Como números não são objetos, você não pode invocar métodos neles. Por exemplo, a chamada `x.sqrt()` nunca será válida em Java.

Eis um exemplo típico de um método estático que realiza alguns cálculos algébricos simples: calcular a porcentagem `p` da quantia `a`. Como os parâmetros são números, o método não opera em absolutamente nenhum objeto, portanto nós o tornamos um método estático:

```
/**
   Calcula uma porcentagem de uma quantia.
   @param p porcentagem a aplicar
   @param a quantia à qual a porcentagem é aplicada
   @return p porcentagem de a
*/
public static double percentOf(double p, double a)
{
   return (p / 100) * a;
}
```

Você precisa encontrar um local para esse método. Vamos pensar em uma nova classe (semelhante à classe `Math` da biblioteca Java padrão). Como o método `percentOf` tem a ver com cálculos financeiros, projetaremos uma classe `Financial` para armazená-lo. Eis a classe:

```
public class Financial
{
   public static double percentOf(double p, double a)
   {
      return (p / 100) * a;
   }
   // Outros métodos financeiros podem ser adicionados aqui.
}
```

Ao chamar um método estático, você fornece o nome da classe que contém o método para que o compilador possa localizá-lo. Por exemplo,

```
double tax = Financial.percentOf(taxRate, total);
```

Observe que você não fornece um objeto do tipo `Financial` ao chamar o método.

Agora podemos dizer por que o método `main` é estático. Quando o programa inicia, não há nenhum objeto. Portanto, o *primeiro* método no programa deve ser um método estático.

Talvez você esteja se perguntando por que esses métodos são chamados estáticos. O significado normal da palavra *estático* ("permanecer fixo em um lugar") não parece estar relacionado com aquilo que os métodos estáticos fazem. Na realidade, essa palavra foi adotada por acidente. Java usa a palavra-chave `static` porque C++ a usa no mesmo contexto. C++ usa `static` para indicar métodos de classe porque os criadores de C++ não queriam criar uma outra palavra-chave. Alguém observou que havia uma palavra-chave

raramente utilizada, `static`, que indica algumas variáveis que permanecem em uma localização fixa para múltiplas chamadas de método. (Java não tem esse recurso, nem precisa dele.) Acabou-se descobrindo que a palavra-chave poderia ser reutilizada para indicar métodos de classe sem confundir o compilador. O fato de que ela pode confundir as pessoas aparentemente não foi uma grande preocupação. Você simplesmente tem de conviver com o fato de que "método estático" significa "método de classe": um método que não opera em um objeto e que só tem parâmetros explícitos.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

12. Suponha que Java não tivesse métodos estáticos. Todos os métodos da classe `Math` seriam então métodos de instância. Como você calcularia a raiz quadrada de x?
13. Harry entrega seu dever de casa, um programa que executa o jogo-da-velha. A solução dele consiste em uma única classe com muitos métodos estáticos. Por que isso não é uma solução orientada a objetos?

8.7 Campos estáticos

Às vezes, você precisa armazenar valores fora de um objeto específico. Utilize campos estáticos para esse propósito. Eis um exemplo típico. Utilizaremos uma versão da nossa classe `BankAccount` em que cada objeto conta bancária tem um saldo e um número de conta:

```
public class BankAccount
{
   . . .
   private double balance;
   private int accountNumber;
}
```

Queremos atribuir números de conta seqüencialmente. Isto é, queremos que o construtor de conta bancária crie a primeira conta com o número 1001, a próxima com o número 1002 e assim por diante. Portanto, devemos armazenar o último número de conta atribuído em algum lugar.

Não faz sentido, porém, transformar esse valor em um campo de instância:

```
public class BankAccount
{
   . . .
   private double balance;
   private int accountNumber;
   private int lastAssignedNumber = 1000; // NÃO – não funcionará
}
```

Nesse caso, cada *instância* da classe `BankAccount` teria um valor próprio de `lastAssignedNumber`.

Um campo estático pertence à classe, não a um objeto da classe.

Em vez disso, precisamos ter um único valor de `lastAssignedNumber` que seja o mesmo para toda a *classe*. Esse campo é chamado campo estático, porque você o declara utilizando a palavra-chave `static`.

```
public class BankAccount
{
   . . .
   private double balance;
   private int accountNumber;
   private static int lastAssignedNumber = 1000;
}
```

Cada objeto `BankAccount` tem campos de instância `balance` e `accountNumber` próprios, mas há apenas uma única cópia da variável `lastAssignedNumber` (veja Figura 4). Esse campo é armazenado em um local separado, fora de qualquer objeto `BankAccount`.

Um campo estático às vezes é chamado *campo de classe* porque há um único campo para toda a classe.

Cada método de uma classe pode acessar seus campos estáticos. Eis o construtor da classe `BankAccount`, que incrementa o último número atribuído e então o usa para inicializar o número de conta do objeto a ser construído:

```
public class BankAccount
{
   public BankAccount()
   {
      // Gera o próximo número de conta a ser atribuído
      lastAssignedNumber++; // Atualiza o campo estático

      // Atribui o campo ao número de conta dessa conta bancária
      accountNumber = lastAssignedNumber; // Configura o campo de instância
   }
   . . .
}
```

Como você inicializa um campo estático? Você não pode configurá-lo no construtor da classe:

```
public BankAccount()
{
   lastAssignedNumber = 1000; // NÃO – seria redefinido para 1000 a cada novo objeto
   . . .
}
```

Assim, a inicialização ocorreria toda vez que uma nova instância fosse construída.

Há três maneiras de inicializar um campo estático:

1. Não fazer nada. O campo estático é então inicializado com `0` (para números), `false` (para valores `boolean`) ou `null` (para objetos).
2. Utilizar um inicializador explícito, como:

```
public class BankAccount
{
   . . .
   private static int lastAssignedNumber = 1000;
}
```

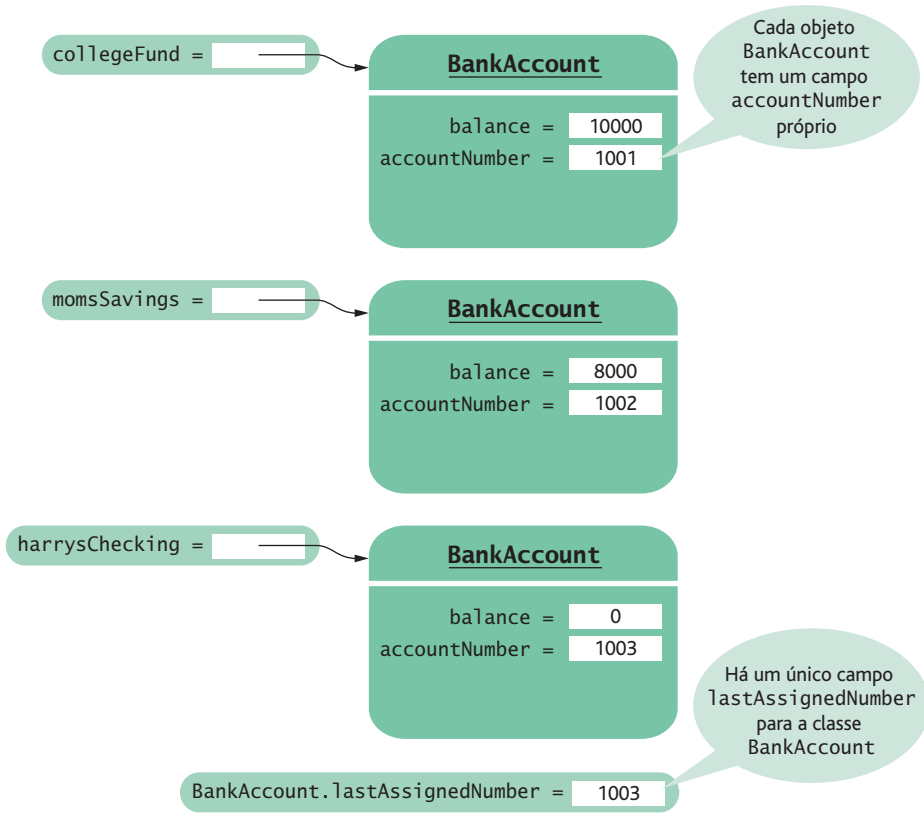


Figura 4 Um campo estático e campos de instância.

A inicialização é executada depois que a classe é carregada.

3. Utilizar um bloco de inicialização estático (ver Tópico Avançado 8.3).

Como ocorre com campos de instância, campos estáticos sempre devem ser declarados como `private` para assegurar que os métodos das outras classes não alterem seus valores. A exceção a essa regra são as *constantes* estáticas, que podem ser privadas ou públicas. Por exemplo, a classe `BankAccount` poderia definir o valor de uma constante pública, como

```
public class BankAccount
{
   . . .
   public static final double OVERDRAFT_FEE = 5;
}
```

Métodos de qualquer classe referenciam essa constante como `BankAccount.OVERDRAFT_FEE`.

Faz sentido declarar constantes como `static` – você não iria querer que cada objeto da classe `BankAccount` tivesse seu próprio conjunto de variáveis com os valores dessas constantes. É suficiente ter um conjunto delas para a classe.

Por que as variáveis de classe são chamadas `static`? Como ocorre com os métodos estáticos, a própria palavra-chave `static` é simplesmente uma remanescente sem sentido

de C++. Mas campos estáticos e métodos estáticos têm muito em comum: eles são aplicados a toda a *classe*, não a instâncias específicas da classe.

Em geral, é recomendável minimizar o uso dos campos e métodos estáticos. Se encontrar utilizando vários métodos estáticos é uma indicação de que talvez você não tenha encontrado as classes corretas para resolver seu problema de uma maneira orientada a objetos.

Autoverificação da aprendizagem

14. Cite dois campos estáticos da classe `System`.
15. Harry informa que encontrou uma excelente maneira de evitar esses objetos incômodos: colocar todo o código em uma única classe e declarar todos os métodos e campos como `static`. Então `main` pode chamar os outros métodos estáticos e todos eles podem acessar os campos estáticos. O plano do Harry funcionará? Ele é uma boa idéia?

Tópico avançado 8.3

Formas alternativas de inicialização de campos

O Tópico Avançado 8.3 abrange dois mecanismos menos comuns para inicialização de campo: especificar os valores iniciais para os campos e usar blocos de inicialização.

8.8 Escopo

8.8.1 Escopo das variáveis locais

O escopo de uma variável é a região de um programa em que a variável pode ser acessada.

Se você tiver múltiplas variáveis ou campos com o mesmo nome, haverá a possibilidade de conflito. Para entender os potenciais problemas, você precisa conhecer o escopo de cada variável: a parte do programa em que a variável pode ser acessada.

O escopo de uma variável local se estende do ponto da sua declaração ao final do bloco que a inclui.

Às vezes acontece que o mesmo nome de variável é utilizado em dois métodos. Considere as variáveis `r` no exemplo a seguir:

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {
      double r = rect.getWidth() * rect.getHeight();
      return r;
   }
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r);
   }
}
```

Essas variáveis são independentes entre si, em outras palavras, seus escopos não são relacionados. Podemos ter variáveis locais com o mesmo nome `r` em métodos diferentes, assim como pode haver diferentes hotéis com o mesmo nome "Bates Motel" em diferentes cidades.

O escopo de uma variável local não pode conter a definição de outra variável com o mesmo nome.

Em Java, o escopo de uma variável local nunca pode conter a definição de outra variável local com o mesmo nome. Por exemplo, o seguinte é um erro:

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
   double r = Math.sqrt(x);
   // Erro – não é permitido declarar uma outra variável
   // chamada r aqui
   . . .
}
```

Entretanto, podemos ter variáveis locais com nomes idênticos se seus escopos não se sobrepuserem, como

```
if (x >= 0)
{
   double r = Math.sqrt(x);
   . . .
} // O escopo de r termina aqui
else
{
   Rectangle r = new Rectangle(5, 10, 20, 30);
   // OK – é válido declarar outro r aqui
   . . .
}
```

8.8.2 Escopo dos membros de classe

Um nome qualificado é prefixado pelo nome da sua classe ou por uma referência a um objeto, como `Math.sqrt` ou `other.balance`.

Nesta seção, consideramos o escopo de campos e métodos de uma classe. (Estes são chamados coletivamente de membros da classe.) Membros privados têm *escopo de classe*: você pode acessar todos os membros em qualquer um dos métodos da classe.

Se quiser utilizar um método ou campo público fora da classe, você deve *qualificar* o nome. Você qualifica um método ou campo estático especificando o nome da classe, como `Math.sqrt` ou `Math.PI`. Você qualifica um campo ou método de instância especificando o objeto ao qual o campo ou método deve ser aplicado, como `harrysChecking.getBalance()`.

Um nome não-qualificado de campo de instância ou de método referencia o parâmetro `this`.

Dentro de um método, você não precisa qualificar campos ou métodos que pertencem à própria classe. Campos de instância referenciam automaticamente o parâmetro implícito do método, isto é, o objeto em que o método é invocado. Por exemplo, considere o método `transfer`:

```
public class BankAccount
{
   public void transfer(double amount, BankAccount other)
   {
      balance = balance - amount; // isto é, this.balance
      other.balance = other.balance + amount;
   }
   . . .
}
```

Aqui, o nome não-qualificado `balance` significa `this.balance`. (Lembre-se do Capítulo 3, `this` é uma referência ao parâmetro implícito de qualquer método.)

A mesma regra se aplica aos métodos. Portanto, uma outra implementação do método `transfer` é

```
public class BankAccount
{
   public void transfer(double amount, BankAccount other)
   {
      withdraw(amount); // isto é,this.withdraw(amount);
      other.deposit(amount);
   }
   . . .
}
```

Sempre que você ver uma chamada de método de instância sem um parâmetro implícito, o método é chamado no parâmetro `this`. Essa chamada de método recebe o nome de "auto-chamada".

Da mesma forma, você pode utilizar um campo ou método estático da mesma classe sem qualificador. Por exemplo, considere a versão a seguir do método `withdraw`:

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (balance < amount) balance = balance - OVERDRAFT_FEE;
      else . . .
   }
   . . .
   private static double OVERDRAFT_FEE = 5;
}
```

Aqui, o nome não-qualificado `OVERDRAFT_FEE` referencia `BankAccount.OVERDRAFT_FEE`.

8.8.3 Escopos sobrepostos

Problemas aparecem se houver dois nomes de variáveis idênticos com escopos que se sobrepõem. Isso nunca pode ocorrer com as variáveis locais, mas os escopos das variáveis locais e dos campos de instância com nomes idênticos podem se sobrepor. Eis um exemplo propositadamente ruim.

```
public class Coin
{
   . . .
   public double getExchangeValue(double exchangeRate)
```

```
   {
      double value; // Variável local
      . . .
      return value;
   }
   private String name;
   private double value; // Campo com o mesmo nome
}
```

Uma variável local pode sombrear um campo com o mesmo nome. Você pode acessar o nome do campo sombreado qualificando-o com a referência `this`.

Dentro do método `getExchangeValue`, o nome da variável `value` poderia ter dois significados: a variável local ou o campo de instância. A linguagem Java especifica que nessa situação a variável *local* tem preferência. Ela *sombreia* o campo de instância. Isso parece bem arbitrário, mas há realmente uma boa razão: você ainda pode referenciar o campo de instância como `this.value`.

```
value = this.value * exchangeRate;
```

Não é necessário escrever código como este: você pode alterar facilmente o nome da variável local para alguma outra coisa, como `result`.

Mas você deve estar ciente de um uso comum da referência `this`. Ao implementar construtores, muitos programadores acham cansativo pensar em nomes diferentes para campos e parâmetros de instância. A referência `this` resolve esse problema. Eis um exemplo típico.

```
public Coin(double value, String name)
{
   this.value = value;
   this.name = name;
}
```

A expressão `this.value` referencia o campo de instância, mas `value` é o parâmetro. Naturalmente, você sempre pode renomear os parâmetros de construção para `aValue` e `aName`, como fizemos neste livro.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

16. Considere o método `deposit` da classe `BankAccount`. Qual é o escopo das variáveis `amount` e `newBalance`?
17. Qual é o escopo do campo `balance` da classe `BankAccount`?

ERRO COMUM 8.2

Sombreamento

O uso acidental do mesmo nome para uma variável local e um campo de instância é um erro incrivelmente comum. Como vimos na seção anterior, a variável local *obscurece* o campo de instância. Ainda que seu objetivo fosse acessar o campo de instância, é a variável local que é silenciosamente acessada. Por alguma razão, esse problema é mais comum nos construtores. Examine este exemplo de um construtor incorreto:

```
public class Coin
{
   public Coin(double aValue, String aName)
   {
      value = aValue;
      String name = aName; // Ops . . .
   }
   . . .
   private double value;
   private String name;
}
```

O programador declarou uma variável local `name` no construtor. É bem provável que isso tenha sido apenas um erro de digitação – os dedos do programador estavam no piloto automático e ele digitou a palavra-chave `String`, embora todo o tempo pretendesse acessar o campo de instância. Infelizmente, o compilador não fornece nenhum alerta nessa situação e configura silenciosamente a variável local com o valor de `aName`. O campo de instância do objeto que está sendo construído nunca é tocado e permanece `null`. Alguns programadores fornecem um prefixo especial para todos os nomes dos campos de instância para diferenciá-los de outras variáveis. Uma convenção comum é prefixar todos os nomes dos campos de instância com o prefixo `my`, como `myValue` ou `myName`.

Dica de produtividade 8.1

Pesquisa e substituição global

A Dica de Produtividade 8.1 discute as opções do recurso "pesquisar e substituir" nos editores avançados de programação.

Dica de produtividade 8.2

Expressões regulares

A Dica de Produtividade 8.2 discute as expressões regulares, uma notação comumente utilizada para especificar padrões de caracteres.

Tópico avançado 8.4

Importações estáticas

O Tópico Avançado 8.4 introduz a sintaxe "static import" para importar constantes estáticas de modo que possam ser utilizadas sem um prefixo de classe.

8.9 Pacotes

8.9.1 Organizando classes relacionadas em pacotes

Um pacote é um conjunto de classes relacionadas.

Um programa Java consiste em uma coleção de classes. Até agora, a maioria dos seus programas consistiu em um pequeno número de classes. Mas, à medida que o tamanho dos programas aumenta, simplesmente distribuir as classes em vários arquivos não é suficiente. Um mecanismo adicional de estruturação é necessário.

Em Java, os pacotes fornecem esse mecanismo de estruturação. Um pacote Java é um conjunto de classes relacionadas. Por exemplo, a biblioteca Java consiste em vários pacotes, alguns dos quais estão listados na Tabela 1.

Para colocar classes em um pacote, você deve adicionar uma linha

`package` *nomeDoPacote*`;`

como a primeira instrução no arquivo-fonte que contém a classe. Um nome de pacote consiste em um ou mais identificadores separados por pontos. (Consulte a Seção 8.9.3 para dicas sobre como criar nomes de pacotes.)

Por exemplo, vamos adicionar a classe `Financial`, introduzida neste capítulo, a um pacote com o nome de `com.horstmann.bigjava`. O arquivo `Financial.java` deve iniciar assim:

```
package com.horstmann.bigjava;

public class Financial
{
   . . .
}
```

Tabela 1 Pacotes importantes na biblioteca Java

Pacote	Propósito	Classe de exemplo
`java.lang`	Suporte da linguagem	`Math`
`java.util`	Utilitários	`Random`
`java.io`	Entrada e saída	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Redes	`Socket`
`java.sql`	Acesso a banco de dados por meio da Structured Query Language	`ResultSet`
`javax.swing`	Interface com o usuário Swing	`JButton`
`omg.org.CORBA`	CORBA (Common Object Request Broker Architecture) para objetos distribuídos	`IntHolder`

SINTAXE 8.2 Especificação de pacote

```
package nomeDoPacote;
```

Exemplo:

```
package com.horstmann.bigjava;
```

Objetivo:

Declarar que todas as classes nesse arquivo pertencem a um pacote particular

Além dos pacotes identificados (como `java.util` ou `com.horstmann.bigjava`), há um pacote especial, chamado *pacote padrão*, que não tem nome. Se você não incluir nenhuma instrução `package` na parte superior do seu arquivo-fonte, as classes serão colocadas no pacote padrão.

8.9.2 Importando pacotes

Se quiser utilizar uma classe de um pacote, você pode referenciá-la pelo nome completo (nome da classe mais o nome do pacote). Por exemplo, `java.util.Scanner` referencia a classe `Scanner` no pacote `java.util`:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

A diretiva `import` permite referenciar uma classe de um pacote pelo nome da classe, sem o prefixo do pacote.

Naturalmente, isso é um pouco inconveniente. Em vez disso, você pode *importar* um nome com uma instrução `import`:

```
import java.util.Scanner;
```

Você pode então referenciar a classe como `Scanner` sem o prefixo do pacote.

Você pode importar *todas as classes* de um pacote com uma instrução `import` que termina em `.*`. Por exemplo, você pode utilizar a instrução

```
import java.util.*;
```

para importar todas as classes do pacote `java.util`. Essa instrução permite referenciar classes como `Scanner` ou `Random` sem um prefixo `java.util`.

Mas você nunca precisa importar explicitamente as classes do pacote `java.lang`. Esse é o pacote que contém as classes Java mais básicas, como `Math` e `Object`. Essas classes sempre estão disponíveis para você. Na realidade, uma instrução `import java.lang.*;` automática foi adicionada a cada arquivo-fonte.

Por fim, você não precisa importar outras classes no mesmo pacote. Por exemplo, ao implementar a classe `homework1.Tester`, você não precisa importar a classe `homework1.Bank`. O compilador encontrará a classe `Bank` sem uma instrução `import` porque ela está localizada no mesmo pacote, `homework1`.

8.9.3 Nomes de pacote

Colocar classes relacionadas em um pacote é claramente um mecanismo conveniente para organizar classes. Entretanto, há uma razão mais importante para os pacotes: evitar *conflitos de nomes.* Em um grande projeto, é inevitável que duas pessoas utilizem o mesmo nome para o mesmo conceito. Isso acontece até mesmo na biblioteca de classes Java padrão (que agora tem milhares de classes). Há uma classe `Timer` no pacote `java.util` e outra classe chamada `Timer` no pacote `javax.swing`. Você ainda pode instruir o compilador Java sobre qual classe `Timer` você precisa simplesmente referenciando-as como `java.util.Timer` e `javax.swing.Timer`.

Naturalmente, para que a convenção de atribuição de nomes a pacotes funcione, deve haver alguma maneira de assegurar que nomes de pacote sejam únicos. Não seria bom se o fabricante de carros BMW colocasse todo o código Java no pacote `bmw` e um outro programador (talvez Bertha M. Walters) tivesse a mesma idéia brilhante. Para evitar esse problema, os criadores de Java recomendam usar um esquema para atribuição de nomes a pacotes que tire proveito da unicidade dos nomes de domínio da Internet.

Utilize um nome de domínio ao contrário para criar nomes de pacote não-ambíguos.

Por exemplo, tenho um nome de domínio `horstmann.com` e não há outra pessoa neste planeta que tem esse mesmo nome de domínio. (Foi pura sorte que o nome de domínio `horstmann.com` não tenha sido escolhido por nenhuma outra pessoa quando o registrei. Se seu nome for Walters, infelizmente você descobrirá que outra pessoa já registrou o domínio para `walters.com`.) Para obter um nome de pacote, escreva o nome de domínio de trás para frente a fim de criar um prefixo do nome do pacote, como `com.horstmann`.

Se você não tiver seu próprio nome de domínio, ainda será possível criar um nome de pacote que tem boa probabilidade de ser único escrevendo seu endereço de email de trás para frente. Por exemplo, se Bertha Walters tiver um endereço de email `walters@cs.sjsu.edu`, você poderá utilizar um nome de pacote `edu.sjsu.cs.walters` para suas próprias classes.

Alguns instrutores vão querer que você coloque cada um dos seus trabalhos de casa em um pacote separado, como `homework1`, `homework2` e assim por diante. A razão é novamente evitar conflitos de nomes. Você pode ter duas classes, `homework1.Bank` e `homework2.Bank`, com propriedades um pouco diferentes.

8.9.4 Como as classes são localizadas

Se o compilador Java estiver configurado adequadamente no seu sistema e se você só utilizar classes padrão, via de regra você não precisará se preocupar com a localização dos arquivos das classes e pode seguramente pular esta seção. Mas se quiser adicionar seus próprios pacotes ou se o compilador não puder localizar uma classe ou pacote particular, você precisará entender o mecanismo.

O caminho de um arquivo de classe deve corresponder ao nome do pacote.

Um pacote está localizado em um subdiretório que corresponde ao seu próprio nome. As partes do nome entre os pontos representam diretórios sucessivamente aninhados. Por exemplo, o pacote `com.horstmann.bigjava` seria armazenado em um subdire-

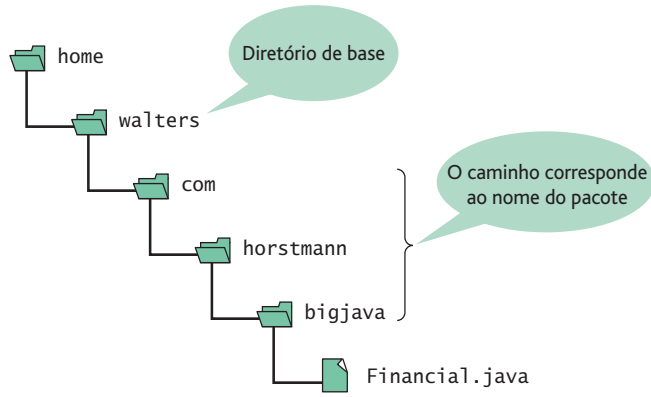


Figura 5
Diretórios e subdiretórios de base para pacotes.

tório `com/horstmann/bigjava`. Se o pacote só for utilizado em conjunto com um único programa, você pode então colocar o subdiretório dentro do diretório que armazena esses arquivos de programa. Por exemplo, se armazenar suas lições de casa em um *diretório de base* `/home/walters`, você então poderá colocar os arquivos de classe para o pacote `com.horstmann.bigjava` no diretório `/home/walters/com/horstmann/bigjava`, como mostrado na Figura 5. (Aqui, estamos utilizando nomes de arquivo no estilo UNIX. No Windows, você poderia utilizar `c:\home\walters\com\horstmann\bigjava`.)

Mas, se quiser colocar seus programas em vários diretórios diferentes, como `/home/walters/hw1`, `/home/walters/hw2`, . . . , provavelmente você não vai querer ter vários subdiretórios idênticos `/home/walters/hw1/com/horstmann/bigjava`, `/home/walters/hw2/com/horstmann/bigjava` e assim por diante. Nesse caso, é recomendável criar um único diretório com um nome como `/home/walters/lib/com/horstmann/bigjava`, colocar todos os arquivos de classe para o pacote nesse diretório e informar o compilador Java de uma vez por todas como localizar os arquivos de classe.

Você precisa adicionar os diretórios que poderiam conter pacotes no *caminho de classe*. No exemplo anterior, você adicionou o diretório `/home/walters/lib` a esse caminho de classe. Os detalhes para fazer isso dependem do seu ambiente de compilação; consulte seu instrutor ou a documentação do seu compilador. Se utilizar o Sun Java SDK, você precisará configurar o caminho de classe. O comando exato depende do sistema operacional. No UNIX, o comando poderia ser:

```
export CLASSPATH=/home/walters/lib;.
```

Essa configuração coloca tanto o diretório `/home/walters/lib` como o diretório atual no caminho de classe. (O ponto indica o diretório atual.)

Um exemplo típico para o Windows seria:

```
set CLASSPATH=c:\home\walters\lib;.
```

Observe que o caminho de classe contém *diretórios de base* que podem conter diretórios de pacote. É um erro comum colocar o endereço completo do pacote no caminho da classe. Se o caminho de classe contiver equivocadamente `/home/walters/lib/com/`

`horstmann/bigjava`, então o compilador tentará localizar o pacote `com.horstmann.bigjava` em `/home/walters/lib/com/horstmann/bigjava/com/horstmann/bigjava` e não localizará os arquivos.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

18. Quais dos seguintes são pacotes?

a. `java`

b. `java.lang`

c. `java.util`

d. `java.lang.Math`

19. Um programa Java sem instruções `import` está limitado ao uso dos pacotes padrão e do pacote `java.lang`?

20. Suponha que suas lições de casa estejam localizadas no diretório `/home/me/cs101` (`c:\me\cs101` no Windows). Seu instrutor fala para você colocar sua lição de casa em pacotes. Em que diretório você coloca a classe `hw1.problem1.TicTacToeTester`?

ERRO COMUM 8.3

Uso confuso de pontos

Em Java, o símbolo de ponto (`.`) é utilizado como um separador nas situações a seguir:

- Entre os nomes dos pacotes (`java.util`)
- Entre os nomes dos pacotes e das classes (`homework1.Bank`)
- Entre os nomes das classes e suas classes internas (`Ellipse2D.Double`)
- Entre os nomes das classes e das variáveis de instância (`Math.PI`)
- Entre objetos e métodos (`account.getBalance()`)

Quando você vê uma longa cadeia de nomes separados por pontos, pode ser muito difícil descobrir qual parte corresponde ao nome do pacote, qual parte é o nome da classe, qual parte é um nome de variável de instância e qual parte é um nome de método. Considere:

```
java.lang.System.out.println(x);
```

Como `println` é seguido por um parêntese de abertura, ele precisa ser um nome de método. Portanto, `out` precisa ser um objeto ou uma classe com um método `println` estático. (Naturalmente, sabemos que `out` é uma referência a um objeto do tipo `PrintStream`.) Novamente, não fica absolutamente claro, sem o contexto, se `System` é um outro objeto, com uma variável pública `out`, ou uma classe com uma variável `static`. Julgando a partir do número de páginas que a especificação da linguagem Java [2] dedica a essa questão, até mesmo o compilador tem problemas para interpretar essas seqüências de strings separadas por pontos.

Para evitar problemas, é útil adotar um estilo estrito de codificação. Se os nomes de classe sempre iniciarem com uma letra maiúscula e os nomes de variável, de método e de pacote sempre iniciarem com uma letra minúscula, essa confusão poderá ser evitada.

COMO FAZER 8.1

Programando com pacotes

Esta seção explica em detalhes como colocar seus programas em pacotes. Por exemplo, talvez seu instrutor solicite que você coloque cada lição de casa em um pacote separado. Dessa maneira, você pode ter classes com o mesmo nome, mas diferentes implementações em pacotes separados (como `homework1.Bank` e `homework2.Bank`).

Passo 1 Crie um nome de pacote.

Seu instrutor talvez forneça um nome de pacote que você pode utilizar, como `homework1`. Ou, talvez você queira utilizar um nome de pacote que seja único. Inicie com seu endereço de email, escrito de trás para frente. Por exemplo, `walters@cs.sjsu.edu` torna-se `edu.sjsu.cs.walters`. Adicione então um subpacote que descreva seu projeto ou lição de casa, como `edu.sjsu.cs.walters.homework1`.

Passo 2 Selecione um *diretório de base*.

O diretório de base é o diretório que contém os diretórios para seus vários pacotes, por exemplo, `/home/walters` ou `c:\cs1`.

Passo 3 Crie um subdiretório a partir do diretório de base que corresponda ao nome do seu pacote.

O subdiretório precisa estar contido no seu diretório de base. Cada segmento deve corresponder a um segmento do nome do pacote. Por exemplo,

```
mkdir /home/walters/homework1
```

Se você tiver múltiplos segmentos, crie-os um por um:

```
mkdir c:\cs1\edu
mkdir c:\cs1\edu\sjsu
mkdir c:\cs1\edu\sjsu\cs
mkdir c:\cs1\edu\sjsu\cs\walters
mkdir c:\cs1\edu\sjsu\cs\walters\homework1
```

Passo 4 Coloque seus arquivos-fonte no subdiretório de pacote.

Por exemplo, se sua lição de casa consiste nos arquivos `Tester.java` e `Bank.java`, coloque-os em:

```
/home/walters/homework1/Tester.java
/home/walters/homework1/Bank.java
```

ou

```
c:\cs1\edu\sjsu\cs\walters\homework1\Tester.java
c:\cs1\edu\sjsu\cs\walters\homework1\Bank.java
```

Passo 5 Utilize a instrução `package` em cada arquivo-fonte.

A primeira linha não-comentário de cada arquivo deve ser uma instrução de pacote que liste o nome do pacote, como:

```
package homework1;
```

ou

```
package edu.sjsu.cs.walters.homework1;
```

Passo 6 Compile seus arquivos-fonte a partir do *diretório de base*.

Passe para o diretório de base (a partir do Passo 2) para compilar seus arquivos. Por exemplo,

```
cd /home/walters
javac homework1/Tester.java
```

ou

```
cd \cs1
javac edu\sjsu\cs\walters\homework1\Tester.java
```

Observe que o compilador Java precisa do *nome do arquivo-fonte e não do nome da classe. Isto é, você precisa fornecer separadores de arquivo* (/ no UNIX, \ no Windows) e uma extensão de arquivo (`.java`).

Passo 7 Execute seu programa a partir do *diretório de base*.

Diferentemente do compilador Java, o interpretador Java precisa do *nome da classe (e não de um nome de arquivo) que contém o método* `main`. Isto é, utilize pontos como separadores de pacote e não utilize uma extensão de arquivo. Por exemplo,

```
cd /home/walters
java homework1.Tester
```

ou

```
cd \cs1
java edu.sjsu.cs.walters.homework1.Tester
```

FATO ALEATÓRIO 8.1

O crescimento explosivo dos computadores pessoais

O Fato Aleatório 8.1 investiga a história do computador pessoal, desde o advento do primeiro microprocessador até o primeiro Macintosh.

8.10 Frameworks de teste de unidade

Até agora, utilizamos uma abordagem muito simples para o teste. Fornecemos classes testadoras cujo método `main` calcula e imprime os valores reais e os esperados. Entretanto, essa abordagem tem duas limitações. Leva tempo inspecionar a saída e decidir se um teste passou. Mais importante, o método `main` se torna confuso se contiver muitos testes.

Frameworks de teste de unidade simplificam a tarefa de escrever classes que contenham vários casos de teste.

Frameworks de teste de unidade foram projetados para executar e avaliar rapidamente conjuntos de teste e facilitar a adição incremental dos casos de teste. Um dos frameworks de teste mais famosos é o JUnit. Ele está livremente disponível em `http://junit.org` e também é incorporado a alguns ambientes de desenvolvimento, incluindo o BlueJ e o Eclipse.

Ao utilizar o JUnit, você projeta uma classe de teste acompanhante para cada classe que desenvolve. Há atualmente duas versões do JUnit em uso comum, a 3 e a 4. Des-

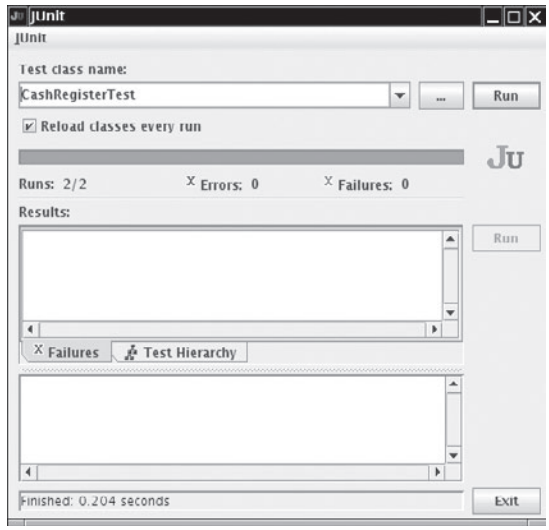


Figura 6
Teste de unidade com o JUnit.

creveremos essas duas versões. No JUnit 3, sua classe de teste tem duas propriedades essenciais:

- A classe de teste deve estender a classe `TestCase` a partir do pacote `junit.framework`.
- Para cada caso de teste, você deve definir um método cujo nome inicia com `test`, como `testSimpleCase`.

Em cada caso de teste, você faz alguns cálculos e então computa alguma condição que você acredita ser verdadeira. Você então passa o resultado para um método, que passa o resultado de um teste ao framework, mais comumente o método `assertEquals`. O método `assertEquals` recebe como parâmetros os valores esperados e os reais e, para números de ponto flutuante, um valor de tolerância.

Também é comum (mas não necessário) que o nome da classe de teste termine em `Test`, como `CashRegisterTest`. Considere este exemplo:

```
import junit.framework.TestCase;

public class CashRegisterTest extends TestCase

{
   public void testSimpleCase()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.enterPayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      assertEquals(expected, register.giveChange(), EPSILON);
   }

   public void testZeroBalance()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(2.25);
      register.recordPurchase(19.25);
```

```
      register.enterPayment(21, 2, 0, 0, 0);
      assertEquals(0, register.giveChange(), EPSILON);
   }
   // Mais casos de teste
   . . .
   private static final double EPSILON = 1E-12;
}
```

Se todos os casos de teste forem bem-sucedidos, a ferramenta JUnit mostrará uma barra verde (veja Figura 6). Se um dos casos de teste falhar, a ferramenta JUnit mostrará uma barra vermelha e uma mensagem de erro.

Sua classe de teste também pode ter outros métodos (cujos nomes não devem iniciar com `test`). Em geral, esses métodos executam passos que você quer compartilhar entre métodos de teste.

O JUnit 4 é ainda mais simples. Sua classe de teste não precisa estender nenhuma classe e você pode escolher livremente nomes para seus métodos de teste. Utilize "anotações" para marcar os métodos de teste. Uma anotação é um recurso avançado do Java que adiciona um marcador ao código que é interpretado por uma outra ferramenta. No caso do JUnit, a anotação `@Test` é utilizada para marcar métodos de teste.

```
import org.junit.Test
import org.junit.Assert;

public class CashRegisterTest
{
   @Test public void simpleCase()
   {
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.enterPayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      Assert.assertEquals(expected, register.giveChange(), EPSILON);
   }
   // Mais casos de teste
   . . .
}
```

A filosofia do JUnit é executar todos os testes sempre que você mudar seu código.

A filosofia do JUnit é simples. Sempre que você implementar uma classe, crie também uma classe de teste acompanhante. Você cria os testes à medida que projeta o programa, um método de teste por vez. A quantidade de casos de teste aumenta continuamente na classe de teste. Sempre que você detectar uma falha real, adicione um caso de teste que a esvazia para que possa certificar-se de que não irá reintroduzir esse erro particular. Sempre que você modifica sua classe, simplesmente execute os testes novamente.

Se todos os testes forem bem-sucedidos, a interface com o usuário exibe uma barra verde e você pode relaxar. Caso contrário, há uma barra vermelha, mas isso também é bom. É muito mais fácil corrigir um erro isoladamente do que dentro de um programa complexo.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

21. Forneça uma classe de teste JUnit com um caso de teste para a classe `Earthquake` do Capítulo 5.

22. Qual é a importância do parâmetro `EPSILON` no método `assertEquals`?

RESUMO DO CAPÍTULO

1. Uma classe deve representar um único conceito do domínio do problema, como negócios, ciência ou matemática.
2. A interface pública de uma classe é coesa se todos os seus recursos estiverem relacionados ao conceito que a classe representa.
3. Uma classe depende de outra se ela usa objetos dessa classe.
4. Uma boa prática é minimizar o acoplamento (isto é, a dependência) entre classes.
5. Uma classe imutável não tem nenhum método modificador.
6. Um efeito colateral de um método é qualquer modificação nos dados que seja observável externamente.
7. Você deve minimizar os efeitos colaterais que extrapolem a modificação do parâmetro implícito.
8. Em Java, um método nunca pode alterar parâmetros de tipos primitivos.
9. Uma pré-condição é um requisito que o chamador de um método deve cumprir. Se um método for chamado violando uma pré-condição, esse método não será responsável por calcular o resultado correto.
10. Uma assertiva é uma condição lógica em um programa que você acredita ser verdade.
11. Se um método foi chamado de acordo com suas pré-condições, então ele precisa assegurar que suas pós-condições são válidas.
12. Um método estático não é invocado em um objeto.
13. Um campo estático pertence à classe, não a um objeto da classe.
14. O escopo de uma variável é a região de um programa em que a variável pode ser acessada.
15. O escopo de uma variável local não pode conter a definição de outra variável com o mesmo nome.
16. Um nome qualificado é prefixado pelo nome da classe ou por uma referência a objeto, como `Math.sqrt` ou `other.balance`.
17. Um nome não-qualificado de um campo ou método de instância referencia o parâmetro `this`.
18. Uma variável local pode sombrear um campo com o mesmo nome. Você pode acessar o nome do campo sombreado qualificando-o com a referência `this`.
19. Um pacote é um conjunto de classes relacionadas.
20. A diretiva `import` permite referenciar uma classe de um pacote pelo nome da classe, sem o prefixo do pacote.
21. Utilize um nome de domínio ao contrário para criar nomes de pacote não-ambíguos.
22. O caminho de um arquivo de classe deve corresponder ao nome do pacote.
23. Frameworks de teste de unidade simplificam a tarefa de escrever classes que contêm vários casos de teste.
24. A filosofia do JUnit é executar todos os testes sempre que você mudar seu código.

LEITURA ADICIONAL

1. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1989, Chapter 7.
2. `http://java.sun.com/docs/books/jls` Especificação da linguagem Java.

EXERCÍCIOS DE REVISÃO

★★ **Exercício R8.1.** Considere a descrição do seguinte problema:

> Os usuários colocam moedas em uma máquina de venda automática e selecionam um produto pressionando um botão. Se as moedas inseridas forem suficientes para cobrir o preço de compra do produto, o produto e o troco são entregues. Caso contrário, as moedas inseridas são devolvidas ao usuário.

Quais classes você deve utilizar para implementá-lo?

★★ **Exercício R8.2.** Considere a descrição do seguinte problema:

> Os funcionários recebem seus cheques de pagamento quinzenalmente. Eles são pagos pelo valor de cada hora trabalhada. Entretanto, se trabalharem mais de 40 horas por semana, eles receberão horas extras com 150% do valor dos seus salários normais.

Quais classes você deve utilizar para implementá-lo?

★★ **Exercício R8.3.** Considere a descrição do seguinte problema:

> Clientes fazem pedidos de produtos em uma loja. Faturas são geradas para listar os itens e as quantias pedidas, pagamentos recebidos e valores devidos. Produtos são despachados para o endereço do cliente e as faturas são enviadas ao endereço de cobrança.

Quais classes você deve utilizar para implementá-lo?

★★★ **Exercício R8.4.** Examine a interface pública da classe `java.lang.System` e discuta se ela é coesa.

★★ **Exercício R8.5.** Suponha que um objeto `Invoice` contenha as descrições dos produtos pedidos e o endereço de cobrança e de remessa do cliente. Desenhe um diagrama UML que mostra as dependências entre as classes `Invoice`, `Address`, `Customer` e `Product`.

★★ **Exercício R8.6.** Suponha que uma máquina de venda automática contenha produtos e que os usuários insiram moedas nessa máquina para comprá-los. Desenhe um diagrama UML que mostre as dependências entre as classes `VendingMachine`, `Coin` e `Product`.

★★ **Exercício R8.7.** De quais classes a classe `Integer` na biblioteca padrão depende?

★★ **Exercício R8.8.** De quais classes a classe `Rectangle` na biblioteca padrão depende?

★ **Exercício R8.9.** Classifique os métodos da classe `Scanner` utilizados neste livro como modificadores e de acesso.

★ **Exercício R8.10.** Classifique os métodos da classe `Rectangle` como modificadores e de acesso.

★ **Exercício R8.11.** Quais classes a seguir são imutáveis?

a. `Rectangle`
b. `String`
c. `Random`

★ **Exercício R8.12.** Quais classes a seguir são imutáveis?

a. `PrintStream`
b. `Date`
c. `Integer`

★★ **Exercício R8.13.** Qual efeito colateral, se há algum, os três métodos a seguir têm:

```
public class Coin
{
   public void print()
   {
      System.out.println(name + " " + value);
   }

   public void print(PrintStream stream)
   {
      stream.println(name + " " + value);
   }

   public String toString()
   {
      return name + " " + value;
   }
   . . .
}
```

★★★ **Exercício R8.14.** Idealmente, um método não deve ter efeito colateral. Você consegue escrever um programa no qual nenhum método tenha efeito colateral? Esse programa seria útil?

★★ **Exercício R8.15.** Escreva pré-condições para os métodos a seguir. Não implemente os métodos.

a. `public static double sqrt(double x)`
b. `public static String romanNumeral(int n)`
c. `public static double slope(Line2D.Double a)`
d. `public static String weekday(int day)`

★★ **Exercício R8.16.** Quais pré-condições os métodos a seguir na biblioteca Java padrão contêm?

a. `Math.sqrt`
b. `Math.tan`

c. `Math.log`

d. `Math.exp`

e. `Math.pow`

f. `Math.abs`

★★ **Exercício R8.17.** Quais pré-condições os métodos a seguir na biblioteca Java padrão contêm?

a. `Integer.parseInt(String s)`

b. `StringTokenizer.nextToken()`

c. `Random.nextInt(int n)`

d. `String.substring(int m, int n)`

★★★ **Exercício R8.18.** Quando um método é chamado com parâmetros que violam a(s) pré-condição(es), ele pode terminar (lançando uma exceção ou um erro de assertiva) ou pode retornar ao chamador. Forneça dois exemplos de métodos de biblioteca (métodos padrão ou métodos de biblioteca utilizados neste livro) que retornem algum resultado aos chamadores quando chamados com parâmetros inválidos e forneça dois exemplos de métodos de biblioteca que terminam.

★★ **Exercício R8.19.** Considere uma classe `CashRegister` com métodos

- `public void enterPayment(int coinCount, Coin coinType)`
- `public double getTotalPayment()`

Forneça uma pós-condição razoável para o método `enterPayment`. Quais pré-condições você precisaria para que a classe `CashRegister` possa assegurar essa pós-condição?

★★ **Exercício R8.20.** Considere o método a seguir concebido para trocar os valores dos dois números de ponto flutuante:

```
public static void falseSwap(double a, double b)
{
   double temp = a;
   a = b;
   b = temp;
}
public static void main(String[] args)
{
   double x = 3;
   double y = 4;
   falseSwap(x, y);
   System.out.println(x + " " + y);
}
```

Por que o método não troca o conteúdo de `x` e `y`?

★★★ **Exercício R8.21.** Como você pode escrever um método que troque dois números de ponto flutuante? *Dica:* `Point2D.Double`.

★★ **Exercício R8.22.** Desenhe um diagrama de memória que mostre por que o método não pode trocar dois objetos `BankAccount`:

```
public static void falseSwap(BankAccount a, BankAccount b)
{
   BankAccount temp = a;
   a = b;
   b = temp;
}
```

★ **Exercício R8.23.** Considere um aprimoramento da classe `Die` do Capítulo 6 com o seguinte campo estático:

```
public class Die
{
   public Die(int s) { . . . }
   public int cast() { . . . }
   private int sides;
   private static Random generator = new Random();
}
```

Desenhe um diagrama de memória que mostre três dados:

```
Die d4 = new Die(4);
Die d6 = new Die(6);
Die d8 = new Die(8);
```

Certifique-se de indicar os valores dos campos `sides` e `generator`.

★ **Exercício R8.24.** Tente compilar o programa a seguir. Explique a mensagem de erro que você recebe.

```
public class Print13
{
   public void print(int x)
   {
      System.out.println(x);
   }
   public static void main(String[] args)
   {
      int n = 13;
      print(n);
   }
}
```

★ **Exercício R8.25.** Examine os métodos da classe `Integer`. Quais são estáticos? Por quê?

★★ **Exercício R8.26.** Examine os métodos da classe `String` (mas ignore aqueles que recebem um parâmetro do tipo `char[]`). Quais são estáticos? Por quê?

★★ **Exercício R8.27.** Os campos `in` e `out` da classe `System` são campos públicos estáticos da classe `System`. Esse é um bom design? Se não for, como você poderia aprimorá-lo?

★★ **Exercício R8.28.** Na classe a seguir, a variável `n` ocorre em múltiplos escopos. Quais declarações de `n` são válidas e quais são inválidas?

```
public class X
{
   public int f()
   {
      int n = 1;
      return n;
   }
   public int g(int k)
   {
      int a;
      for (int n = 1; n <= k; n++)
         a = a + n;
      return a;
   }
   public int h(int n)
   {
      int b;
      for (int n = 1; n <= 10; n++)
         b = b + n;
      return b + n;
   }
   public int k(int n)
   {
      if (n < 0)
      {
         int k = -n;
         int n = (int) (Math.sqrt(k));
         return n;
      }
      else return n;
   }
   public int m(int k)
   {
      int a;
      for (int n = 1; n <= k; n++)
         a = a + n;
      for (int n = k; n >= 1; n++)
         a = a + n;
      return a;
   }
   private int n;
}
```

★ **Exercício R8.29.** O que é um nome qualificado? O que é um nome não-qualificado?

★★ **Exercício R8.30.** Quando você acessa um nome não-qualificado em um método, o que esse acesso significa? Discuta os significados dos termos "instância" e "estático".

★★ **Exercício R8.31.** Todo programa Java pode ser reescrito para evitar instruções `import`. Explique como e reescreva `RectangleComponent.java` do Capítulo 2 para evitar instruções `import`.

★ **Exercício R8.32.** O que é o pacote padrão? Você já tinha utilizado esse pacote antes deste capítulo na sua programação?

★★T **Exercício R8.33.** O que JUnit faz quando um método de teste lança uma exceção? Teste-o e informe seus resultados.

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★★ **Exercício P8.1.** Implemente a classe `Coin` descrita na Seção 8.2. Modifique a classe `CashRegister` para que moedas possam ser adicionadas à caixa registradora, fornecendo o método

```
void enterPayment(int coinCount, Coin coinType)
```

O chamador precisa invocar esse método diversas vezes, uma para cada tipo de moeda presente no pagamento.

★★ **Exercício P8.2.** Modifique o método `giveChange` da classe `CashRegister` para que ele retorne o número de moedas de um tipo particular a retornar:

```
int giveChange(Coin coinType)
```

O chamador precisa invocar esse método para cada tipo de moeda, em valores decrescentes.

★ **Exercício P8.3.** Caixas registradoras reais podem tratar cédulas e moedas. Projete uma única classe que expresse a semelhança desses conceitos. Reprojete a classe `CashRegister` e forneça um método para inserir pagamentos que são descritos pela sua classe. Seu principal desafio é pensar em um bom nome para essa classe.

★ **Exercício P8.4.** Aprimore a classe `BankAccount` adicionando pré-condições ao construtor e ao método `deposit` que exijam que o parâmetro `amount` seja pelo menos zero e uma pré-condição para o método `withdraw` que exige que `amount` tenha um valor entre 0 e o saldo atual. Utilize assertivas para testar as pré-condições.

★★ **Exercício P8.5.** Escreva os métodos estáticos

- `public static double sphereVolume(double r)`
- `public static double sphereSurface(double r)`
- `public static double cylinderVolume(double r, double h)`
- `public static double cylinderSurface(double r, double h)`
- `public static double coneVolume(double r, double h)`
- `public static double coneSurface(double r, double h)`

que calculam o volume e a área de superfície de uma esfera com raio `r`, um cilindro com uma base circular com raio `r` e altura `h`, e um cone com uma base circular com raio `r` e altura `h`. Coloque-os em uma classe `Geometry`. Escreva então um programa que solicita ao usuário os valores de `r` e `h`, chame os seis métodos e imprima os resultados.

★ **Exercício P8.6.** Resolva o Exercício P8.5 implementando as classes `Sphere`, `Cylinder` e `Cone`. Qual abordagem é mais orientada a objetos?

★ **Exercício P8.7.** Escreva os métodos

```
public static double perimeter(Ellipse2D.Double e)
public static double area(Ellipse2D.Double e)
```

que calculam a área e o perímetro da elipse `e`. Adicione esses métodos a uma classe `Geometry`. A parte mais difícil desse exercício é encontrar e implementar uma equação exata para o perímetro. Por que faz sentido utilizar um método estático nesse caso?

★★ **Exercício P8.8.** Escreva os métodos

```
public static double angle(Point2D.Double p, Point2D.Double q)
public static double slope(Point2D.Double p, Point2D.Double q)
```

que calculem o ângulo entre o eixo x e a linha que une dois pontos, medidos em graus e o declive dessa linha. Adicione os métodos à classe `Geometry`. Forneça pré-condições adequadas. Por que faz sentido utilizar um método estático nesse caso?

★★ **Exercício P8.9.** Escreva os métodos

```
public static boolean isInside(Point2D.Double p, Ellipse2D.Double e)
public static boolean isOnBoundary(Point2D.Double p, Ellipse2D.Double e)
```

que testam se um ponto está dentro ou no limite de uma elipse. Adicione os métodos à classe `Geometry`.

★ **Exercício P8.10.** Escreva o método

```
public static int readInt(
      Scanner in, String prompt, String error, int min, int max)
```

que exibe a string no prompt, lê um inteiro e testa se ele está entre o mínimo e o máximo. Se não estiver, imprime uma mensagem de erro e repete a leitura da entrada. Adicione o método a uma classe `Input`.

★★ **Exercício P8.11.** Considere o algoritmo a seguir para calcular x^n para um inteiro n. Se $n < 0$, x^n é $1/x^{-n}$. Se n for positivo e par, então $x^n = (x^{n/2})^2$. Se n for positivo e ímpar, então $x^n = x^{n-1} \cdot x$. Implemente um método estático `double intPower(double x, int n)` que usa esse algoritmo. Adicione-o a uma classe chamada `Numeric`.

★★ **Exercício P8.12.** Aprimore a classe `Needle` do Capítulo 6. Transforme o campo `generator` em um campo estático para que todas as agulhas compartilhem um único gerador de números aleatórios.

★★ **Exercício P8.13.** Implemente uma classe `Coin` e uma classe `CashRegister` como descrito no Exercício P8.1. Coloque as classes em um pacote chamado `money`. Mantenha a classe `CashRegisterTester` no pacote padrão.

★ **Exercício P8.14.** Coloque uma classe `BankAccount` em um pacote cujo nome seja baseado no seu endereço de email, como descrito na Seção 8.9. Mantenha a classe `BankAccountTester` no pacote padrão.

★★T **Exercício P8.15.** Forneça uma classe de teste JUnit `BankTest` com três métodos de teste, cada um dos quais testa um método diferente da classe `Bank` do Capítulo 7.

★★T **Exercício P8.16.** Forneça uma classe de teste JUnit `TaxReturnTest` com três métodos de teste que testam situações diferentes de imposto para a classe `Tax` do Capítulo 5.

★G **Exercício P8.17.** Escreva os métodos

- `public static void drawH(Graphics2D g2, Point2D.Double p);`
- `public static void drawE(Graphics2D g2, Point2D.Double p);`
- `public static void drawL(Graphics2D g2, Point2D.Double p);`
- `public static void drawO(Graphics2D g2, Point2D.Double p);`

que mostram as letras H, E, L, O na janela gráfica, onde o ponto `p` é o canto superior esquerdo da letra. Chame então os métodos para desenhar as palavras "HELLO" e "HOLE" na tela gráfica. Desenhe linhas e elipses. Não utilize o método `drawString`. Não utilize `System.out`.

★★G **Exercício P8.18.** Repita o Exercício P8.15 projetando as classes `LetterH`, `LetterE`, `LetterL` e `LetterO`, cada uma com um construtor que recebe um parâmetro `Point2D.Double` (o canto superior esquerdo) e um método `draw(Graphics2D g2)`. Qual solução é mais orientada a objetos?

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 8.1.** Implemente um programa que imprime cheques de pagamento para um grupo de estudantes assistentes. Subtraia os impostos federais e de contribuição previdenciária. (Talvez você queira utilizar o cálculo do imposto utilizado no Capítulo 5. Pesquise na Internet as contribuições da previdência social.) Seu programa deve solicitar os nomes, salários por hora e horas trabalhadas de cada estudante.

★★★ **Projeto 8.2.** Para classificar mais rapidamente as cartas, o serviço postal norte-americano encoraja empresas que enviam grandes volumes de correio a utilizar um código de barras para indicar o CEP (veja Figura 7).

Figura 7 Código de barras postal.

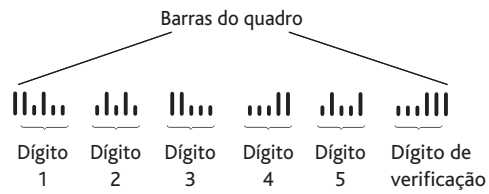


Figura 8 Codificação para códigos de barras de cinco dígitos.

O esquema de codificação para um CEP de cinco dígitos é mostrado na Figura 8. Há uma barra de identificação com altura máxima de cada lado.

Os cinco dígitos codificados são seguidos por um dígito de verificação, que é calculado assim: some todos os dígitos e escolha o dígito de verificação para tornar a soma um múltiplo de 10. Por exemplo, a soma dos dígitos no CEP 95014 é 19, portanto o dígito de verificação é 1 para tornar a soma igual a 20.

Cada dígito do CEP, e o dígito de verificação, é codificado de acordo com esta tabela:

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

onde 0 indica uma meia barra e 1 uma barra completa. Observe que elas representam todas as combinações de duas barras completas e três meias barras. O dígito pode ser calculado facilmente a partir do código de barras utilizando os pesos de coluna 7, 4, 2, 1, 0. Por exemplo, 01100 é:

$$0 \cdot 7 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + 0 \cdot 0 = 6$$

A única exceção é 0, que produziria 11 de acordo com a fórmula do peso.

Escreva um programa que solicita ao usuário um CEP e imprime o código de barras correspondente. Utilize `:` para meias barras e `|` para barras cheias. Por exemplo, 95014 torna-se:

```
||:|:::|:|:||::::::||:|::|:::|||
```

(Alternativamente, escreva uma aplicação gráfica que desenha barras reais.)

Seu programa também deve ser capaz de executar a conversão oposta: converter barras em seus códigos de CEP, informando qualquer erro no formato de entrada ou uma não-correspondência dos dígitos.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Procure substantivos na descrição do problema.
2. Sim (`ChessBoard`) e não (`MovePiece`).

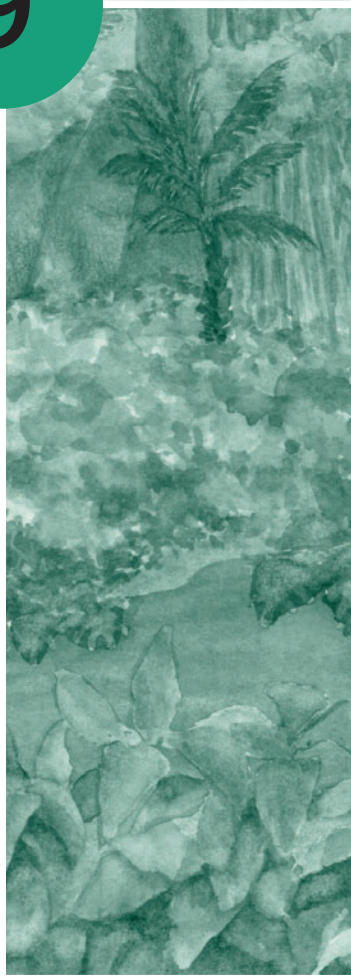
3. Alguns dos seus recursos lidam com pagamentos, outros com valores de moeda.
4. Nenhuma das operações com moeda requer a classe `CashRegister`.
5. Se uma classe não depende de outra, ela não é afetada pelas alterações de interface na outra classe.
6. É um método de acesso – chamar `substring` não modifica a string em que o método é invocado. De fato, todos os métodos da classe `String` são métodos de acesso.
7. Não – `translate` é um método modificador.
8. É um efeito colateral; esse tipo de efeito é comum na programação orientada a objetos.
9. Sim – o método afeta o estado do parâmetro `Scanner`.
10. Assim você não precisa se preocupar com a verificação de valores inválidos – isso se torna responsabilidade do chamador.
11. Não – você pode executar qualquer ação que seja conveniente para você.
12. `Math m = new Math(); y = m.sqrt(x);`
13. Em uma solução orientada a objetos, o método `main` construiria objetos das classes `Game`, `Player`, etc. A maioria dos métodos seriam métodos de instância que dependem do estado desses objetos.
14. `System.in` e `System.out`.
15. Sim, funciona. Métodos estáticos podem acessar campos estáticos da mesma classe. Mas isso é uma idéia ruim. À medida que suas tarefas de programação tornam-se mais complexas, é recomendável utilizar objetos e classes para organizar seus programas.
16. O escopo de `amount` é todo o método `deposit`. O escopo de `newBalance` inicia no ponto em que a variável é definida e se estende até o fim do método.
17. Inicia no começo da classe e termina no final da classe.
18. (a) Não; (b) Sim; (c) Sim; (d) Não
19. Não – você simplesmente usa nomes completamente qualificados para todas as outras classes, como `java.util.Random` e `java.awt.Rectangle`.
20. `/home/me/cs101/hw1/problem1` ou, no Windows, `c:\me\cs101\hw1\problem1`.
21. Eis uma possível resposta, utilizando o estilo do JUnit 4.

```
public class EarthquakeTest
{
   @Test public void testLevel4()
   {
      Earthquake quake = new Earthquake(4);
      Assert.assertEquals("Felt by many people, no destruction",
            quake.getDescription());
    }
}
```

22. É um limiar de tolerância para comparar números de ponto flutuante. Queremos que o teste de igualdade passe se houver um pequeno erro de arredondamento.

Capítulo 9

Interfaces e Polimorfismo



Objetivos do capítulo

- Aprender sobre interfaces
- Ser capaz de converter entre referências da classe e referências da interface
- Entender o conceito de polimorfismo
- Entender como interfaces podem ser utilizadas para desacoplar classes
- Aprender a implementar classes auxiliares como classes internas
- Entender como classes internas acessam variáveis do escopo que as circunda
- G Implementar ouvintes de eventos em aplicações gráficas

Para aumentar a produtividade em programação, queremos ser capazes de *reutilizar* componentes de software em múltiplos projetos. Mas algumas adaptações freqüentemente são necessárias para tornar a reutilização possível. Neste capítulo, você aprenderá uma estratégia importante para separar a parte reutilizável de um cálculo das partes que variam em cada cenário da reutilização. A parte reutilizável invoca métodos de uma *interface*. Ela é combinada com uma classe que implementa os métodos da interface. Para produzir uma aplicação diferente, você simplesmente adiciona outra classe que implementa os mesmos métodos. O comportamento do programa varia de acordo com a classe que foi adicionada – esse fenômeno é chamado de *polimorfismo*.

Conteúdo do Capítulo

9.1 Utilizando interfaces para reutilização de código 360
Sintaxe 9.1: Definindo uma interface 364
Sintaxe 9.2: Implementando uma interface 364
Erro comum 9.1: Esquecendo de definir a implementação de métodos como públicos 366
Tópico avançado 9.1: Constantes em interfaces 366

9.2 Convertendo entre classes e tipos interface 367
Erro comum 9.2: Tentando instanciar uma interface 368

9.3 Polimorfismo 368

9.4 Utilizando interfaces para retornos de chamada 370

9.5 Classes internas 375
Sintaxe 9.3: Classes internas 375
Tópico avançado 9.2: Classes anônimas ⊕
Fato aleatório 9.1: Sistemas operacionais ⊕

9.6G Eventos, fontes de eventos e ouvintes de eventos 377
Erro comum 9.3: Modificando a assinatura na implementação do método 380

9.7G Utilizando classes internas para ouvintes 381

9.8G Construindo aplicações com botões 384
Erro comum 9.4: Esquecendo de relacionar um ouvinte 387
Dica de produtividade 9.1: Não utilize um contêiner como um ouvinte 387

9.9G Processando eventos de temporizador ⊕
Erro comum 9.5: Esquecendo de repintar 388

9.10G Eventos de mouse ⊕
Tópico avançado 9.3: Adaptadores de evento ⊕
Fato aleatório 9.2: Linguagens de programação ⊕

9.1 Utilizando interfaces para reutilização de código

Utilize tipos interface para tornar o código mais reutilizável.

Às vezes é possível tornar o código mais geral e reutilizável focando nas operações essenciais que são executadas. *Tipos interface* são utilizados para expressar essas operações comuns.

Considere a classe `DataSet` do Capítulo 6. Utilizamos essa classe para calcular os valores médio e máximo de um conjunto de valores de entrada. Mas essa classe só é adequada para calcular a média de um conjunto de *números*. Se quiséssemos processar contas bancárias a fim de localizar a conta bancária com o saldo mais alto, teríamos de modificar a classe, assim:

```
public class DataSet // Modificada para objetos da classe BankAccount
{
   . . .
   public void add(BankAccount x)
   {
      sum = sum + x.getBalance();
      if (count == 0 || maximum.getBalance() < x.getBalance())
         maximum = x;
      count++;
   }
   public BankAccount getMaximum()
   {
      return maximum;
   }
```

```
   private double sum;
   private BankAccount maximum;
   private int count;
}
```

Ou suponha que quiséssemos encontrar a moeda com o valor mais alto entre uma série de moedas. Precisaríamos modificar a classe `DataSet` novamente.

```
public class DataSet // Modificada para objetos da classe Coin
{
   . . .
   public void add(Coin x)
   {
      sum = sum + x.getValue();
      if (count == 0  || maximum.getValue() < x.getValue())
         maximum = x;
      count++;
   }
   public Coin getMaximum()
   {
      return maximum;
   }
   private double sum;
   private Coin maximum;
   private int count;
}
```

Claramente, a mecânica fundamental da análise de dados é a mesma em todos os casos, mas os detalhes da medição diferem.

Suponha que as várias classes concordassem em ter um único método `getMeasure` para obter a medida a ser utilizada na análise dos dados. Para contas bancárias, `getMeasure` retorna o saldo. Para moedas, `getMeasure` retorna o valor da moeda e assim por diante. Poderíamos então implementar uma única classe `DataSet` reutilizável, cujo método `add` se parece com isto:

```
sum = sum + x.getMeasure();
if (count == 0 || maximum.getMeasure() < x.getMeasure())
   maximum = x;
count++;
```

Qual é o tipo da variável `x`? Idealmente, `x` deve referenciar qualquer classe que tenha um método `getMeasure`.

Um tipo interface em Java declara um conjunto de métodos e suas assinaturas.

Em Java, um tipo interface é utilizado para especificar as operações requeridas. Definiremos um tipo interface chamado `Measurable`:

```
public interface Measurable
{
   double getMeasure();
}
```

A declaração da interface lista todos os métodos que o tipo interface requer. O tipo interface `Measurable` requer um único método, mas, em geral, um tipo interface pode exigir diversos métodos.

Observe que o tipo `Measurable` não é um tipo da biblioteca padrão – ele é um tipo criado especificamente para este livro a fim de tornar a classe `DataSet` mais reutilizável.

Diferentemente de uma classe, um tipo interface não fornece nenhuma implementação.

Um tipo interface é semelhante a uma classe, mas há várias diferenças importantes:

- Todos os métodos em um tipo interface são abstratos; isto é, eles têm um nome, parâmetros e um tipo de retorno, mas não têm uma implementação.
- Todos os métodos em um tipo interface são automaticamente públicos.
- Um tipo interface não tem campos de instância.

Agora podemos utilizar o tipo interface `Measurable` para declarar as variáveis `x` e `maximum`.

```
public class DataSet
{
   . . .
   public void add(Measurable x)
   {
      sum = sum + x.getMeasure();
      if (count == 0  || maximum.getMeasure() < x.getMeasure())
         maximum = x;
      count++;
   }
   public Measurable getMaximum()
   {
      return maximum;
   }
   private double sum;
   private Measurable maximum;
   private int count;
}
```

Utilize a palavra-chave `implements` para indicar que uma classe implementa um tipo interface.

Essa classe `DataSet` pode ser utilizada para analisar objetos de qualquer classe que implemente a interface `Measurable`. Uma classe implementa um tipo interface se ela declarar a interface em uma cláusula `implements`. Ela deve então implementar o método ou os métodos que a interface requer.

```
class NomeDaClasse implements Measurable
{
   public double getMeasure()
   {
      Implementação
   }
   Métodos e campos adicionais
}
```

Uma classe pode implementar mais de um tipo interface. Naturalmente, a classe deve então definir todos os métodos que são requeridos por todas as interfaces que ela implementa.

Vamos modificar a classe `BankAccount` para implementar a interface `Measurable`.

```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

Observe que essa classe deve declarar o método como `public`, enquanto a interface não – todos os métodos em uma interface são públicos.

De maneira semelhante, é fácil modificar a classe `Coin` para implementar a interface `Measurable`.

```
public class Coin implements Measurable
{
   public double getMeasure()
   {
      return value;
   }
   . . .
}
```

Resumindo, a interface `Measurable` expressa o que todos os objetos mensuráveis têm em comum. Essa semelhança torna a classe `DataSet` reutilizável. Os objetos da classe `DataSet` podem ser utilizados para analisar coleções de objetos de *qualquer* classe que implemente a interface `Measurable`. A seguir mostramos um programa de teste que ilustra esse fato.

Interfaces podem reduzir o acoplamento entre classes.

A Figura 1 mostra os relacionamentos entre a interface `Measurable`, as classes que implementam a interface e a classe `DataSet` que usa essa interface. Na notação UML, interfaces são marcadas com um indicador de "estereótipo" `«interface»`. Uma seta ponti-

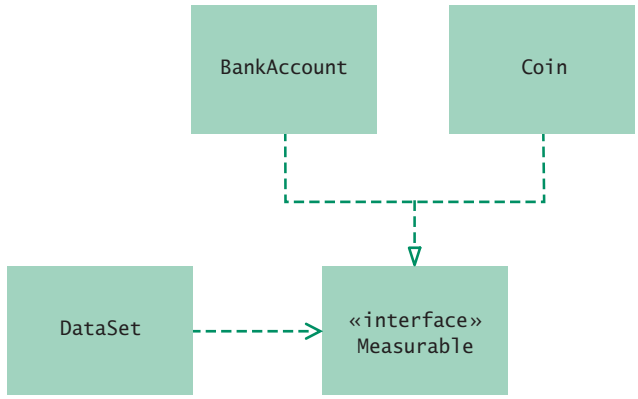


Figura 1 Diagrama UML da classe `DataSet` e as classes que implementam a interface `Measurable`.

lhada com uma ponta triangular indica o relacionamento "*é um*" entre uma classe e uma interface. Você precisa examinar cuidadosamente as pontas das setas – uma linha pontilhada com uma ponta de seta aberta (>) indica o relacionamento ou dependência "*usa*".

Esse diagrama mostra que a classe `DataSet` só depende da interface `Measurable`. Ela é desacoplada das classes `BankAccount` e `Coin`.

SINTAXE 9.1 Definindo uma interface

```
public interface NomeDaInterface
{
   assinaturas de métodos
}
```

Exemplo:

```
public interface Measurable
{
   double getMeasure();
}
```

Objetivo:

Definir uma interface e as assinaturas de seus métodos. Os métodos são automaticamente públicos.

SINTAXE 9.2 Implementando uma interface

```
public class NomeDaClasse
      implements NomeDaInterface, NomeDaInterface, . . .
{
   métodos
   campos
}
```

Exemplo:

```
public class BankAccount implements Measurable
{
   // Outros métodos de BankAccount
   public double getMeasure()
   {
      // Implementação do método
   }
}
```

Objetivo:

Definir uma nova classe que implementa os métodos de uma interface

ch09/measure1/DataSetTester.java

```
/**
   Este programa testa a classe DataSet.
*/
public class DataSetTester
{
   public static void main(String[] args)
   {
      DataSet bankData = new DataSet();

      bankData.add(new BankAccount(0));
      bankData.add(new BankAccount(10000));
      bankData.add(new BankAccount(2000));

      System.out.println("Average balance: "
            + bankData.getAverage());
      System.out.println("Expected: 4000");
      Measurable max = bankData.getMaximum();
      System.out.println("Highest balance: "
            + max.getMeasure());
      System.out.println("Expected: 10000");

      DataSet coinData = new DataSet();

      coinData.add(new Coin(0.25, "quarter"));
      coinData.add(new Coin(0.1, "dime"));
      coinData.add(new Coin(0.05, "nickel"));

      System.out.println("Average coin value: "
            + coinData.getAverage());
      System.out.println("Expected: 0.133");
      max = coinData.getMaximum();
      System.out.println("Highest coin value: "
            + max.getMeasure());
      System.out.println("Expected: 0.25");
   }
}
```

Saída

```
Average balance: 4000.0
Expected: 4000
Highest balance: 10000.0
Expected: 10000
Average coin value: 0.13333333333333333
Expected: 0.133
Highest coin value: 0.25
Expected: 0.25
```

Autoverificação da aprendizagem

1. Suponha que você queira utilizar a classe `DataSet` para localizar o objeto `Country` com a maior população. Qual condição a classe `Country` precisa cumprir?
2. Por que o método `add` da classe `DataSet` não pode ter um parâmetro do tipo `Object`?

Erro comum 9.1

Esquecendo de definir a implementação de métodos como públicos

Os métodos em uma interface não são declarados como `public`, porque eles são públicos por padrão. Mas os métodos em uma classe não são públicos por padrão – o nível de acesso padrão é o acesso de "pacote", que discutiremos no Capítulo 10. É um erro comum esquecer da palavra-chave `public` ao definir um método de uma interface:

```
public class BankAccount implements Measurable
{
   double getMeasure() // Ops – deveria ser: public
   {
      return balance;
   }
   . . .
}
```

O compilador reclama que o método tem um nível de acesso mais fraco, a saber, o acesso de pacote em vez do acesso público. A solução é declarar o método como público.

Tópico avançado 9.1

Constantes em interfaces

Interfaces não podem ter campos de instância, mas é válido especificar *constantes*. Por exemplo, a interface `SwingConstants` define várias constantes, como `SwingConstants.NORTH`, `SwingConstants.EAST`, etc.

Ao definir uma constante em uma interface, você pode (e deve) omitir as palavras-chave `public static final`, porque todos os campos em uma interface são automaticamente `public static final`. Por exemplo,

```
public interface SwingConstants
{
   int NORTH = 1;
   int NORTHEAST = 2;
   int EAST = 3;
   . . .
}
```

9.2 Convertendo entre classes e tipos interface

Interfaces são utilizadas para expressar a semelhança entre as classes. Nesta seção, discutiremos quando é válido converter entre classes e tipos interface.

Examine mais minuciosamente a chamada

```
bankData.add(new BankAccount(10000));
```

a partir do programa de teste da seção anterior. Aqui passamos um objeto do tipo `BankAccount` para o método `add` da classe `DataSet`. Mas esse método tem um parâmetro do tipo `Measurable`:

```
public void add(Measurable x)
```

É válido converter do tipo `BankAccount` para o tipo `Measurable`?

Você pode converter de um tipo classe para um tipo interface, desde que a classe implemente a interface.

Em Java, esse tipo de conversão é válida. Você pode converter de um tipo classe para o tipo interface qualquer que essa classe implementa. Por exemplo,

```
BankAccount account = new BankAccount(10000);
Measurable x = account; // OK
```

Alternativamente, `x` pode referenciar um objeto `Coin`, desde que a classe `Coin` tenha sido modificada para implementar a interface `Measurable`.

```
Coin dime = new Coin(0.1, "dime");
Measurable x = dime; // Também OK
```

Portanto, quando há uma variável de objeto do tipo `Measurable`, na verdade isso quer dizer que você não conhece o tipo exato do objeto que `x` referencia. Tudo o que você sabe é que o objeto tem um método `getMeasure`.

Mas você não pode converter entre tipos não-relacionados:

```
Measurable x = new Rectangle(5, 10, 20, 30); // Erro
```

Essa atribuição é um erro, porque a classe `Rectangle` não implementa a interface `Measurable`.

Ocasionalmente, precisamos converter um objeto em uma referência da interface e então converter esta em um objeto. Isso acontece no método `getMaximum` da classe `DataSet`. O `DataSet` armazena o objeto com a maior medida, *como uma referência* a `Measurable`.

```
DataSet coinData = new DataSet();
coinData.add(new Coin(0.25, "quarter"));
coinData.add(new Coin(0.1, "dime"));
coinData.add(new Coin(0.05, "nickel"));
Measurable max = coinData.getMaximum();
```

Agora, o que você pode fazer com a referência `max`? *Você* sabe que ela referencia um objeto `Coin`, mas o compilador não. Por exemplo, você não pode chamar o método `getName`:

```
String coinName = max.getName(); // Erro
```

Essa chamada é um erro, porque o tipo `Measurable` não tem um método `getName`.

Você precisa fazer uma coerção, ou typecasting, para converter entre um tipo interface e um tipo classe.

Mas, contanto que você esteja absolutamente certo de que `max` referencia um objeto `Coin`, você pode utilizar a notação de coerção para convertê-lo de volta:

```
Coin maxCoin = (Coin) max;
String name = maxCoin.getName();
```

Se você estiver errado e o objeto não referenciar uma moeda, o programa lançará uma exceção e terminará.

Essa notação de coerção é a mesma notação que vimos no Capítulo 4 para converter entre tipos numéricos. Por exemplo, se `x` for um número de ponto flutuante, então `(int) x` será a parte inteira do número. A intenção é semelhante – converter entre um tipo e outro. Entretanto, há uma grande diferença entre coerção de tipos numéricos e coerção de tipos classe. Na coerção, ou typecasting, de tipos numéricos, você *perde informações* e usa a coerção para dizer ao compilador que você concorda com essa perda de informações. Por outro lado, na coerção de tipos objeto, você *assume o risco* de que isso pode resultar em uma exceção e diz ao compilador que você concorda com esse risco.

Autoverificação da aprendizagem

3. Você pode utilizar uma coerção `(BankAccount) x` para converter uma variável `Measurable x` em uma referência `BankAccount`?
4. Se tanto `BankAccount` como `Coin` implementarem a interface `Measurable`, uma referência a `Coin` poderá ser convertida em uma referência a `BankAccount`?

Erro comum 9.2

Tentando instanciar uma interface

Você pode definir variáveis cujo tipo é uma interface, por exemplo:

```
Measurable x;
```

Entretanto, você *nunca* pode criar uma interface:

```
Measurable x = new Measurable(); // Erro
```

Interfaces não são classes. Não há nenhum objeto cujo tipo seja interface. Se uma variável de interface referenciar um objeto, esse objeto deverá pertencer a alguma classe – uma classe que implementa a interface:

```
Measurable x = new BankAccount(); // OK
```

9.3 Polimorfismo

Quando múltiplas classes implementam a mesma interface, cada classe implementa os métodos da interface de diferentes maneiras. Qual é o método correto executado quando o método de interface é invocado? Responderemos essa pergunta nesta seção.

Vale enfatizar mais uma vez que é perfeitamente válido – e de fato muito comum – ter variáveis cujos tipos são uma interface, como

```
Measurable x;
```

Apenas lembre-se de que o objeto que `x` referencia não é do tipo `Measurable`. Na realidade, *nenhum objeto* é do tipo `Measurable`. Em vez disso, o objeto é do tipo de alguma classe que implementa a interface `Measurable`, como `BankAccount` ou `Coin`.

Observe que `x` pode referenciar objetos de *diferentes* tipos durante seu tempo de vida. Aqui a variável `x` primeiro contém uma referência a uma conta bancária e depois uma referência a uma moeda.

```
x = new BankAccount(10000); // OK
x = new Coin(0.1, "dime"); // OK
```

O que você pode fazer com uma variável de interface, já que você não conhece a classe do objeto que ela referencia? Você pode invocar os métodos da interface:

```
double m = x.getMeasure();
```

A classe `DataSet` tirou proveito dessa capacidade calculando a medida do objeto adicionado, sem se preocupar com o tipo exato de objeto que foi adicionado.

Agora vamos pensar na chamada ao método `getMeasure` mais cuidadosamente. *Qual* método `getMeasure`? As classes `BankAccount` e `Coin` fornecem duas implementações *diferentes* desse método. Como o método correto foi chamado se o chamador nem mesmo conhecia a classe exata a que `x` pertence?

A máquina virtual Java faz um esforço especial para localizar o método correto que pertence à classe do objeto real. Isto é, se `x` referenciar um objeto `BankAccount`, o método `BankAccount.getMeasure` será chamado. Se `x` referenciar um objeto `Coin`, o método `Coin.getMeasure` será chamado. Isso significa que uma das chamadas de método

```
double m = x.getMeasure();
```

pode chamar diferentes métodos dependendo do conteúdo momentâneo de `x`.

O polimorfismo indica o princípio de que o comportamento pode variar com base no tipo real de um objeto.

O princípio de que o tipo real do objeto determina o método a ser chamado chama-se polimorfismo. O termo "polimorfismo" vem da palavra grega para "muitas formas". O mesmo cálculo funciona para objetos de muitas formas e se adapta à natureza dos objetos. Em Java, todos os métodos de instância são polimórficos.

Quando você vê uma chamada de método polimórfico, como `x.getMeasure()`, há vários possíveis métodos `getMeasure` que podem ser chamados. Já vimos um outro caso em que o mesmo nome de método pode referenciar diferentes métodos, especialmente quando um nome de método é sobrecarregado: isto é, quando uma única classe tem vários métodos com o mesmo nome, mas tipos diferentes de parâmetro. Por exemplo, você pode ter dois construtores `BankAccount()` e `BankAccount(double)`. O compilador seleciona o método apropriado ao compilar o programa simplesmente examinando os tipos dos parâmetros:

```
account = new BankAccount();
   // O compilador seleciona BankAccount()
account = new BankAccount(10000);
   // O compilador seleciona BankAccount(double)
```

A vinculação inicial dos métodos ocorre quando o compilador selecionar um método entre vários possíveis candidatos. A vinculação tardia ocorre quando a seleção do método acontecer durante a execução do programa.

Há uma diferença importante entre polimorfismo e sobrecarga. O compilador seleciona um método sobrecarregado ao converter o programa, antes de o programa ser executado. Essa seleção de método é chamada vinculação inicial. Entretanto, ao selecionar o método `getMeasure` apropriado em uma chamada `x.getMeasure()`, o compilador não tomou nenhuma decisão ao converter o método. O programa precisa ser executado para que se saiba o que está armazenado em `x`. Portanto, a máquina virtual, não o compilador, seleciona o método apropriado. Essa seleção de método é chamada vinculação tardia.

Autoverificação da aprendizagem

5. Por que é impossível construir um objeto `Measurable`?
6. Por que você pode, apesar disso, declarar uma variável do tipo `Measurable`?
7. O que a sobrecarga e o polimorfismo têm em comum? Onde eles diferem?

9.4 Utilizando interfaces para retornos de chamada

Nesta seção, discutiremos como a classe `DataSet` pode tornar-se ainda mais reutilizável fornecendo um tipo diferente de interface. Esse tipo de interface fornece um mecanismo de "retorno de chamada" (callback), permitindo que a classe `DataSet` chame de volta um método específico quando ela precisar de informações adicionais.

Para entender por que é desejável um outro aprimoramento à classe `DataSet`, considere estas limitações da interface `Measurable`:

- Você só pode adicionar a interface `Measurable` às classes sob seu controle. Se quiser processar um conjunto de objetos `Rectangle`, você não pode fazer a classe `Rectangle` implementar uma outra interface – ela é uma classe do sistema que não pode ser alterada.
- Você pode medir um objeto de uma única maneira. Se você quiser analisar um conjunto de contas de poupança tanto pelo saldo bancário como pela taxa de juros, suas opções são limitadas.

Portanto, vamos repensar a classe `DataSet`. O conjunto de dados precisa ser capaz de medir os objetos que são adicionados. Quando os objetos precisam ser do tipo `Measurable`, a responsabilidade pela medição é dos próprios objetos adicionados, o que é a causa das limitações já observadas. Seria melhor se outro objeto pudesse executar a medição. Vamos mover o método de medição para uma interface diferente:

```
public interface Measurer
{
   double measure(Object anObject);
}
```

O método `measure` mede um objeto e retorna sua medida. Aqui utilizamos o fato de que todos os objetos podem ser convertidos no tipo `Object`, o "mínimo denominador comum" de todas as classes Java. Discutiremos o tipo `Object` mais detalhadamente no Capítulo 10.

A classe `DataSet` aprimorada é construída com um objeto `Measurer` (isto é, um objeto de alguma classe que implementa a interface `Measurer`). Esse objeto é salvo em um campo de instância `measurer` e utilizado para executar medições como esta:

```
public void add(Object x)
{
   sum = sum + measurer.measure(x);
   if (count == 0 || measurer.measure(maximum) < measurer.measure(x))
      maximum = x;
   count++;
}
```

A classe `DataSet` simplesmente cria um retorno de chamada (callback) para o método `measure` sempre que ela precisar medir um objeto.

Agora você pode definir medidores que sejam responsáveis por qualquer tipo de medição. Por exemplo, eis como você pode medir a área de retângulos. Defina uma classe

```
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() * aRectangle.getHeight();
      return area;
   }
}
```

Observe que o método `measure` precisa aceitar um parâmetro do tipo `Object`, ainda que esse medidor específico só meça retângulos. A assinatura do método deve corresponder à assinatura do método `measure` na interface `Measurer`. Então, é feita a coerção do parâmetro `Object` para o tipo `Rectangle`:

```
Rectangle aRectangle = (Rectangle) anObject;
```

O que você pode fazer com um `RectangleMeasurer`? Ele é necessário para um `DataSet` que compara retângulos pela área. Construa um objeto da classe `RectangleMeasurer` e passe-o para o construtor `DataSet`.

```
Measurer m = new RectangleMeasurer();
DataSet data = new DataSet(m);
```

Em seguida, adicione retângulos ao conjunto de dados.

```
data.add(new Rectangle(5, 10, 20, 30));
data.add(new Rectangle(10, 20, 30, 40));
. . .
```

O conjunto de dados solicitará que o objeto `RectangleMeasurer` meça os retângulos. Em outras palavras, o conjunto de dados usa o objeto `RectangleMeasurer` para executar retornos de chamada.

A Figura 2 mostra o diagrama UML das classes e interfaces dessa solução. Como na Figura 1, a classe `DataSet` é desacoplada da classe `Rectangle`, cujos objetos ela processa. Entretanto, ao contrário da Figura 1, a classe `Rectangle` não está mais acoplada com uma outra classe. Em vez disso, para processar retângulos, você tem de pensar em uma pequena classe "auxiliar", `RectangleMeasurer`. Essa classe auxiliar só tem um objetivo: instruir o `DataSet` sobre como medir seus objetos.

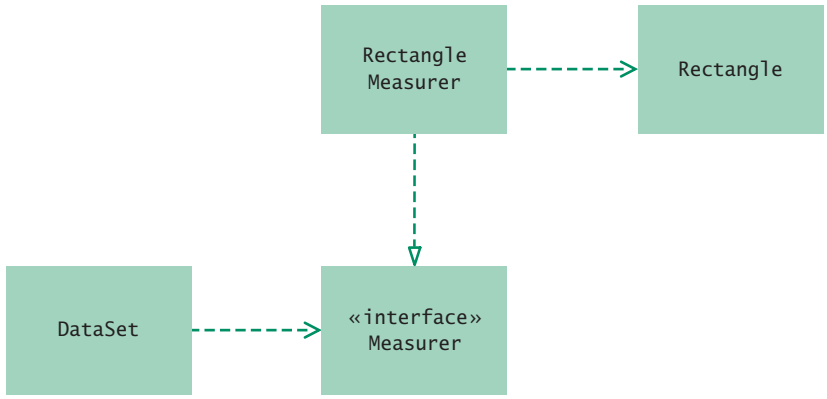


Figura 2 Diagrama UML da classe `DataSet` e da interface `Measurer`.

ch09/measure2/DataSet.java

```
/**
   Calcula a média de um conjunto de valores de dados.
*/
public class DataSet
{
   /**
      Constrói um conjunto de dados vazio com um dado medidor.
      @param aMeasurer medidor utilizado para medir os valores dos dados
   */
   public DataSet(Measurer aMeasurer)
   {
      sum = 0;
      count = 0;
      maximum = null;
      measurer = aMeasurer;
   }

   /**
      Adiciona um valor de dados ao conjunto de dados.
      @param x valor de dados
   */
   public void add(Object x)
   {
      sum = sum + measurer.measure(x);
      if (count == 0
            || measurer.measure(maximum) < measurer.measure(x))
         maximum = x;
      count++;
   }

```

```
31      /**
32         Obtém a média dos dados adicionados.
33         @return média ou 0 se nenhum dado foi adicionado
34      */
35      public double getAverage()
36      {
37         if (count == 0) return 0;
38         else return sum / count;
39      }
40
41      /**
42         Obtém o maior dos dados adicionados.
43         @return máximo ou 0 se nenhum dado foi adicionado
44      */
45      public Object getMaximum()
46      {
47         return maximum;
48      }
49
50      private double sum;
51      private Object maximum;
52      private int count;
53      private Measurer measurer;
54   }
```

ch09/measure2/DataSetTester2.java

```
 1   import java.awt.Rectangle;
 2
 3   /**
 4      Esse programa demonstra o uso de um Measurer.
 5   */
 6   public class DataSetTester2
 7   {
 8      public static void main(String[] args)
 9      {
10         Measurer m = new RectangleMeasurer();
11
12         DataSet data = new DataSet(m);
13
14         data.add(new Rectangle(5, 10, 20, 30));
15         data.add(new Rectangle(10, 20, 30, 40));
16         data.add(new Rectangle(20, 30, 5, 15));
17
18         System.out.println("Average area: " + data.getAverage());
19         System.out.println("Expected: 625");
20
21         Rectangle max = (Rectangle) data.getMaximum();
22         System.out.println("Maximum area rectangle: " + max);
23         System.out.println("Expected: java.awt.Rectangle[
24               x=10,y=20,width=30,height=40]");
25      }
26   }
```

ch09/measure2/Measurer.java

```
/**
   Descreve qualquer classe cujos objetos podem medir outros objetos.
*/
public interface Measurer
{
   /**
      Calcula a medida de um objeto.
      @param anObject objeto a ser medido
      @return medida
   */
   double measure(Object anObject);
}
```

ch09/measure2/RectangleMeasurer.java

```
import java.awt.Rectangle;

/**
   Objetos dessa classe medem retângulos pela área.
*/
public class RectangleMeasurer implements Measurer
{
   public double measure(Object anObject)
   {
      Rectangle aRectangle = (Rectangle) anObject;
      double area = aRectangle.getWidth() * aRectangle.getHeight();
      return area;
   }
}
```

Saída

```
Average area: 625
Expected: 625
Maximum area rectangle: java.awt.Rectangle[x=10,y=20,width=30,height=40]
Expected: java.awt.Rectangle[x=10,y=20,width=30,height=40]
```

Autoverificação da aprendizagem

8. Suponha que você queira utilizar a classe `DataSet` da Seção 9.1 para encontrar a `String` mais longa em um conjunto de entradas. Por que isso não funciona?
9. Como você pode utilizar a classe `DataSet` desta seção para localizar a `String` mais longa em um conjunto de entradas?
10. Por que o método `measure` da interface `Measurer` tem um parâmetro a mais do que o método `getMeasure` da interface `Measurable`?

9.5 Classes internas

A classe `RectangleMeasurer` é uma classe bem trivial. Essas classes só são necessárias porque a classe `DataSet` precisa de um objeto de alguma classe que implemente a interface `Measurer`. Quando você tem uma classe que serve a um propósito bem limitado, como esse, você pode declarar essa classe dentro do método que precisa dela:

```
public class DataSetTester3
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      Measurer m = new RectangleMeasurer();
      DataSet data = new DataSet(m);
      . . .
   }
}
```

Uma classe interna é declarada dentro de outra classe. Classes internas são comumente utilizadas para classes táticas que não devem ser visíveis a partir de outras partes de um programa.

Essa classe é chamada *classe interna.* Uma classe interna é qualquer classe definida dentro de outra classe. Esse arranjo sinaliza ao leitor do seu programa que a classe `RectangleMeasurer` só é importante dentro do escopo desse método. Como uma classe interna dentro de um método não é um recurso publicamente acessível, você não precisa documentá-la integralmente.

Você também pode definir uma classe interna dentro de uma classe que a inclui, mas fora dos métodos. Essa classe interna permanece disponível para todos os métodos da classe que a inclui.

SINTAXE 9.3 Classes internas

Declarada dentro de um método:

```
class NomeDaClasseExterna
{
   assinatura de método
   {
      . . .
      class NomeDaClasseExterna
      {
         métodos
         campos
      }
      . . .
   }
   . . .
}
```

Declarada dentro da classe:

```
class NomeDaClasseExterna
{
   métodos
   campos
   especificadorDeAcesso class
   NomeDaClasseExterna
   {
      métodos
      campos
   }
   . . .
}
```

Exemplo:

```
public class Tester
{
   public static void main(String[] args)
   {
      class RectangleMeasurer implements Measurer
      {
         . . .
      }
      . . .
   }
}
```

Objetivo:

Definir uma classe interna cujo escopo é restrito a um único método ou a métodos de uma única classe

Quando você compilar os arquivos-fonte para um programa que usa as classes internas, examine os arquivos de classe no diretório do seu programa – você descobrirá que as classes internas são armazenadas nos arquivos com nomes curiosos, como `DataSetTester3$1$RectangleMeasurer.class`. Os nomes exatos não são importantes. O fato é que o compilador transforma uma classe interna em um arquivo normal de classe.

ch09/measure3/DataSetTester3.java

```
1   import java.awt.Rectangle;
2
3   /**
4      Esse programa demonstra o uso de uma classe interna.
5   */
6   public class DataSetTester3
7   {
8      public static void main(String[] args)
9      {
10        class RectangleMeasurer implements Measurer
11        {
12           public double measure(Object anObject)
13           {
14              Rectangle aRectangle = (Rectangle) anObject;
15              double area
16                    = aRectangle.getWidth() * aRectangle.getHeight();
17              return area;
18           }
19        }
20
21        Measurer m = new RectangleMeasurer();
22
23        DataSet data = new DataSet(m);
24
25        data.add(new Rectangle(5, 10, 20, 30));
```

```
26        data.add(new Rectangle(10, 20, 30, 40));
27        data.add(new Rectangle(20, 30, 5, 15));
28
29        System.out.println("Average area: " + data.getAverage());
30        System.out.println("Expected: 625");
31
32        Rectangle max = (Rectangle) data.getMaximum();
33        System.out.println("Maximum area rectangle: " + max);
34        System.out.println("Expected: java.awt.Rectangle[
35              x=10,y=20,width=30,height=40]");
36     }
37  }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

11. Por que você utilizaria uma classe interna em vez de uma classe normal?
12. Quantos arquivos de classe são produzidos quando você compila o programa `DataSetTester3`?

TÓPICO AVANÇADO 9.2

Classes anônimas

O Tópico Avançado 9.2 mostra como você pode simplificar a declaração das classes internas com a sintaxe "classe anônima".

FATO ALEATÓRIO 9.1

Sistemas operacionais

O Fato Aleatório 9.1 discute sistemas operacionais, o software que fornece serviços comuns para todos os programas executados em um computador.

9.6 Eventos, fontes de eventos e ouvintes de eventos

Nas aplicações que você escreveu até agora, a entrada de usuário estava sob controle do *programa*. O programa solicitou ao usuário a entrada em uma ordem específica. Por exemplo, um programa poderia solicitar para o usuário fornecer primeiro um nome e então uma quantia em dólares. Mas os programas que você costuma utilizar no seu computador não funcionam dessa maneira. Em um programa com uma interface gráfica moderna, o *usuário* está no controle. O usuário pode utilizar tanto o mouse como o teclado e manipular muitas partes da interface em qualquer ordem desejada. Por exemplo, o usuário pode inserir informações nos campos de texto, ativar menus, clicar em botões e arrastar barras de rolagem em qualquer ordem. O programa deve reagir aos comandos do usuário, independentemente da ordem em que eles chegam. Ter de lidar com muitas entradas possíveis em uma ordem aleatória é bem mais difícil do que simplesmente forçar o usuário a fornecer a entrada em uma ordem fixa.

Eventos da interface com o usuário incluem o pressionamento de uma tecla, movimentos de mouse, cliques em botões, seleções de menu e assim por diante.

Nas seções a seguir, você aprenderá a escrever programas Java que podem reagir a eventos da interface com o usuário, como pressionamentos de botões e cliques de mouse. O conjunto de ferramentas do sistema de janelas Java tem um mecanismo muito sofisticado que permite que um programa especifique os eventos em que ele está interessado e quais objetos deve notificar quando um desses eventos ocorre.

Sempre que o usuário de um programa gráfico digita caracteres ou usa o mouse em qualquer lugar dentro de uma das janelas do programa, o gerenciador de janelas Java envia uma notificação ao programa de que um *evento* ocorreu. O gerenciador de janelas gera uma grande quantidade de eventos. Por exemplo, sempre que ocorre qualquer leve movimento do mouse em uma janela, um evento de "movimento de mouse" é gerado. Eventos também são gerados quando o usuário pressiona uma tecla, clica em um botão ou seleciona um item de menu.

Um ouvinte de evento pertence a uma classe que é fornecida pelo programador da aplicação. Os métodos desse ouvinte descrevem as ações a serem tomadas quando um evento ocorre.

A maioria dos programas não quer ser inundado por eventos cansativos. Por exemplo, quando um botão é clicado com o mouse, o mouse move-se para cima do botão, o botão do mouse é pressionado e por fim o botão é relaxado. Em vez de receber uma grande quantidade de eventos de mouse pouco importantes, um programa pode indicar que ele só se preocupa com cliques de botão, não com os eventos de mouse subjacentes. Mas, se a entrada de mouse for utilizada para desenhar formas em uma tela de pintura virtual, é necessário monitorar detalhadamente os eventos de mouse.

Cada programa deve indicar quais eventos ele precisa receber. O programa faz isso instalando objetos ouvintes de eventos. Um objeto ouvinte de evento pertence a uma classe que você define. Os métodos das suas classes ouvintes de evento contêm as instruções que você quer que sejam executadas quando os eventos ocorrem.

Fontes de eventos informam sobre eventos. Quando um evento ocorre, a fonte do evento notifica todos os ouvintes do evento.

Para instalar um ouvinte, você precisa conhecer a fonte do evento. A fonte do evento é o componente de interface com o usuário que gera um evento específico. Você adiciona um objeto ouvinte de evento às fontes de evento apropriadas. Sempre que o evento ocorre, a fonte do evento chama os métodos apropriados de todos os ouvintes de evento anexados.

Utilize componentes `JButton` para botões. Anexe um `ActionListener` a cada botão.

Isso parece um pouco abstrato, portanto, vamos executar um programa extremamente simples que imprime uma mensagem sempre que um botão é clicado. Ouvintes de botão devem pertencer a uma classe que implementa a interface `ActionListener`:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

Essa interface particular tem um único método, `actionPerformed`. É seu trabalho fornecer uma classe cujo método `actionPerformed` contém as instruções que você quer executar sempre que o botão é clicado. Eis um exemplo muito simples de uma classe ouvinte:

ch09/button1/ClickListener.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3
4  /**
5     Um ouvinte de ação que imprime uma mensagem.
6  */
7  public class ClickListener implements ActionListener
8  {
9     public void actionPerformed(ActionEvent event)
10    {
11       System.out.println("I was clicked.");
12    }
13 }
```

Ignoramos o parâmetro `event` do método `actionPerformed` – ele contém detalhes adicionais sobre o evento, como a data/hora em que ele ocorreu.

Depois de a classe ouvinte ser definida, precisamos construir um objeto dessa classe e adicioná-la ao botão:

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

Sempre que o botão é clicado, ela chama

```
listener.actionPerformed(event);
```

Como resultado, a mensagem é impressa.

Você pode pensar no método `actionPerformed` como um outro exemplo de um retorno de chamada, semelhante ao método `measure` da classe `Measurer`. O conjunto de ferramentas do sistema de janelas chama o método `actionPerformed` sempre que o botão é pressionado, enquanto o `DataSet` chama o método `measure` sempre que ele precisa medir um objeto.

Você pode testar esse programa abrindo uma janela de console, iniciando o programa `ButtonViewer` a partir dessa janela de console, clicando no botão e observando as mensagens na janela de console (veja Figura 3).

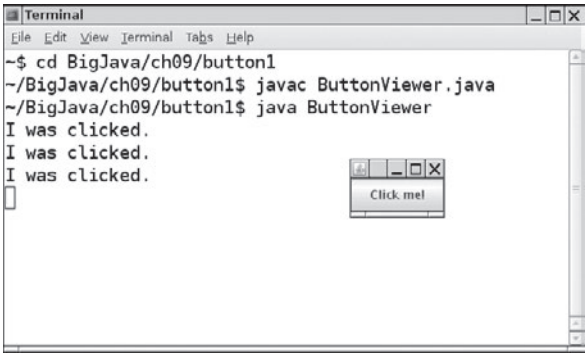


Figura 3 Implementando um ouvinte de ação.

ch09/button1/ButtonViewer.java

```
1  import java.awt.event.ActionListener;
2  import javax.swing.JButton;
3  import javax.swing.JFrame;
4
5  /**
6     Esse programa demonstra como instalar um ouvinte de ação.
7  */
8  public class ButtonViewer
9  {
10    public static void main(String[] args)
11    {
12       JFrame frame = new JFrame();
13       JButton button = new JButton("Click me!");
14       frame.add(button);
15
16       ActionListener listener = new ClickListener();
17       button.addActionListener(listener);
18
19       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
20       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21       frame.setVisible(true);
22    }
23
24    private static final int FRAME_WIDTH = 100;
25    private static final int FRAME_HEIGHT = 60;
26 }
```

Autoverificação da aprendizagem

13. Quais objetos são fontes de eventos e qual é o ouvinte de eventos no programa `ButtonViewer`?
14. Por que é válido atribuir um objeto `ClickListener` a uma variável do tipo `ActionListener`?

Erro comum 9.3

Modificando a assinatura na implementação do método

Ao implementar uma interface, você deve definir cada método *exatamente* como ele foi especificado na interface. Fazer pequenas alterações acidentais nos parâmetros ou nos tipos de retorno é um erro comum. Eis o exemplo clássico,

```
class MyListener implements ActionListener
{
   public void actionPerformed()
   // Ops . . . esqueceu o parâmetro ActionEvent
   {
      . . .
```

```
    }
}
```

No que diz respeito ao compilador, essa classe tem dois métodos:

```
public void actionPerformed(ActionEvent event)
public void actionPerformed()
```

O primeiro método é indefinido. O compilador reclamará que o método não está presente. Você tem de ler a mensagem de erro cuidadosamente e prestar atenção aos parâmetros e aos tipos de retorno para localizar o erro.

9.7 Utilizando classes internas para ouvintes

Na seção anterior, vimos como o código, que é executado quando um botão é clicado, é adicionado a uma classe ouvinte. É comum implementar classes ouvintes como classes internas assim:

```
JButton button = new JButton(". . .");
// Essa classe interna é declarada no mesmo método da variável de botão
class MyListener implements ActionListener
{
   . . .
};

ActionListener listener = new MyListener();
button.addActionListener(listener);
```

Há duas razões para esse arranjo. Primeiro, ele coloca a classe ouvinte trivial exatamente onde ela é necessária, sem poluir o restante do projeto. Além disso, classes internas têm um recurso muito interessante: seus métodos podem acessar variáveis que são definidas nos blocos adjacentes. Nesse sentido, definições de métodos das classes internas comportam-se de maneira semelhante a blocos aninhados.

Lembre-se de que um bloco é um grupo de instruções incluído entre chaves. Se um bloco estiver aninhado dentro de um outro, o bloco interno terá acesso a todas as variáveis do bloco adjacente:

```
{  // Bloco adjacente
   BankAccount account = new BankAccount();
   if (. . .)
   {  // Bloco interno
      . . .
      // OK para acessar variável a partir do bloco adjacente
      account.deposit(interest);
     . . .
   }  // Fim do bloco interno
   . . .
}  // Fim do bloco adjacente
```

Métodos de uma classe interna podem acessar variáveis locais dos blocos adjacentes e campos da classe adjacente.

O mesmo aninhamento funciona para classes internas. Exceto para algumas restrições técnicas que examinaremos mais adiante nesta seção, os métodos de uma classe interna podem acessar as variáveis a partir do escopo que as inclui. Esse recurso é muito útil ao implementar rotinas de tratamento de evento. Ele permite que a classe interna acesse variáveis sem precisar passá-las como parâmetros de método ou de construtor.

Vejamos um exemplo. Suponha que queiramos adicionar juros a uma conta bancária sempre que um botão for clicado.

```
JButton button = new JButton("Add Interest");
final BankAccount account = new BankAccount(INITIAL_BALANCE);

// Essa classe interna é declarada no mesmo método das variáveis account e button.
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // O método ouvinte acessa a variável account
      // do bloco adjacente
      double interest = account.getBalance()
            * INTEREST_RATE / 100;
      account.deposit(interest);
   }
};

ActionListener listener = new AddInterestListener();
button.addActionListener(listener);
```

Variáveis locais que são acessadas por um método de classe interna devem ser declaradas como `final`.

Há um truque técnico. Uma classe interna só pode acessar variáveis *locais adjacentes* se elas forem declaradas como `final`. Isso parece uma restrição, mas normalmente não é um problema. Tenha em mente que uma variável de objeto é `final` sempre que referencia o mesmo objeto. O estado do objeto pode mudar, mas a variável não pode referenciar um objeto diferente. Por exemplo, no nosso programa, nunca houve a pretensão de a variável `account` referenciar múltiplas contas bancárias, portanto não há problema em declará-la como `final`.

Uma classe interna também pode acessar os campos da classe adjacente, novamente com uma restrição. O campo deve pertencer ao objeto que construiu o objeto da classe interna. Se o objeto da classe interna foi criado dentro de um método estático, ele só poderá acessar os campos estáticos adjacentes.

Eis o código-fonte para o programa.

ch09/button2/InvestmentViewer1.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5
```

```
/**
   Esse programa demonstra como um ouvinte de ação pode acessar
   uma variável a partir de um bloco adjacente.
*/
public class InvestmentViewer1
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      // Botão para desencadear o cálculo
      JButton button = new JButton("Add Interest");
      frame.add(button);

      // A aplicação adiciona juros a essa conta bancária
      final BankAccount account = new BankAccount(INITIAL_BALANCE);

      class AddInterestListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            // O método ouvinte acessa a variável account
            // do bloco adjacente
            double interest = account.getBalance()
                  * INTEREST_RATE / 100;
            account.deposit(interest);
            System.out.println("balance: " + account.getBalance());
         }
      }

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }

   private static final double INTEREST_RATE = 10;
   private static final double INITIAL_BALANCE = 1000;

   private static final int FRAME_WIDTH = 120;
   private static final int FRAME_HEIGHT = 60;
}
```

Saída

```
balance: 1100.0
balance: 1210.0
balance: 1331.0
balance: 1464.1
```

Autoverificação da aprendizagem

15. Por que um método da classe interna acessaria uma variável de um escopo adjacente?
16. Se uma classe interna acessar uma variável local de um escopo adjacente, qual regra especial é aplicada?

9.8 Construindo aplicações com botões

Nesta seção, você aprenderá a estruturar uma aplicação gráfica que contém botões. Colocaremos um botão em funcionamento no nosso programa visualizador de investimentos simples. Sempre que o botão é clicado, juros são adicionados a uma conta bancária e o novo saldo é exibido (veja Figura 4).

Primeiro, construímos um objeto da classe `JButton`. Passe o rótulo do botão para o construtor:

```
JButton button = new JButton("Add Interest");
```

Também precisamos de um componente da interface com o usuário que exibe uma mensagem, a saber, os saldos bancários atuais. Esse componente é chamado *rótulo* (ou label). Você passa a string da mensagem inicial para o construtor `JLabel`, da seguinte maneira:

```
JLabel label = new JLabel("balance: " + account.getBalance());
```

> Utilize um contêiner `JPanel` para agrupar múltiplos componentes de interface com o usuário.

O frame da nossa aplicação contém o botão e o rótulo. Mas não podemos simplesmente adicionar os dois componentes diretamente ao frame – eles seriam posicionados um sobre o outro. A solução é colocá-los em um *painel*, um contêiner para outros componentes de interface com o usuário, e então adicionar o painel ao frame:

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Agora estamos prontos para a parte difícil – o ouvinte de evento que trata os cliques de botão. Como na seção anterior, é necessário definir uma classe que implementa a interface `ActionListener` e adicionar a ação de botão ao método `actionPerformed`. Nossa classe ouvinte adiciona juros e exibe o novo saldo:

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double interest = account.getBalance() * INTEREST_RATE / 100;
      account.deposit(interest);
      label.setText("balance: " + account.getBalance());
   }
}
```

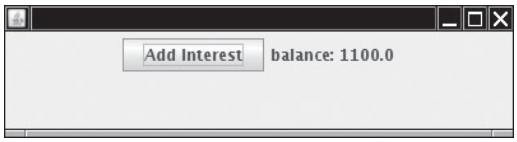


Figura 4 Aplicação com um botão.

Freqüentemente você instala ouvintes de evento como classes internas para que eles possam ter acesso aos campos adjacentes, métodos e variáveis finais.

Há apenas um detalhe técnico menor. O método `actionPerformed` manipula as variáveis `account` e `label`. Ambas são variáveis locais do método `main` do programa visualizador de investimentos, não dos campos de instância da classe `AddInterestListener`. Portanto, precisamos declarar as variáveis `account` e `label` como `final` para que o método `actionPerformed` possa acessá-las.

Vamos montar as partes.

```
public static void main(String[] args)
{
   . . .
   JButton button = new JButton("Add Interest");
   final BankAccount account = new BankAccount(INITIAL_BALANCE);
   final JLabel label = new JLabel("balance: " + account.getBalance());

   class AddInterestListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         double interest = account.getBalance()
               * INTEREST_RATE / 100;
         account.deposit(interest);
         label.setText("balance: " + account.getBalance());
      }
   }

   ActionListener listener = new AddInterestListener();
   button.addActionListener(listener);
   . . .
}
```

Com um pouco de prática, você aprenderá a examinar rapidamente esse código e convertê-lo em português simples: "Quando o botão é clicado, adicione juros e configure o texto do rótulo".

Eis o programa completo. Ele demonstra como adicionar múltiplos componentes a um frame utilizando um painel e como implementar ouvintes como classes internas.

ch09/button3/InvestmentViewer2.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
```

```
6  import javax.swing.JPanel;
7  import javax.swing.JTextField;
8
9  /**
10     Esse programa exibe o rendimento de um investimento.
11 */
12 public class InvestmentViewer2
13 {
14    public static void main(String[] args)
15    {
16       JFrame frame = new JFrame();
17
18       // Botão para desencadear o cálculo
19       JButton button = new JButton("Add Interest");
20
21       // A aplicação adiciona juros a essa conta bancária
22       final BankAccount account = new BankAccount(INITIAL_BALANCE);
23
24       // Rótulo para exibir os resultados
25       final JLabel label = new JLabel(
26             "balance: " + account.getBalance());
27
28       // Painel que contém os componentes da interface com o usuário
29       JPanel panel = new JPanel();
30       panel.add(button);
31       panel.add(label);
32       frame.add(panel);
33
34       class AddInterestListener implements ActionListener
35       {
36          public void actionPerformed(ActionEvent event)
37          {
38             double interest = account.getBalance()
39                   * INTEREST_RATE / 100;
40             account.deposit(interest);
41             label.setText(
42                   "balance: " + account.getBalance());
43          }
44       }
45
46       ActionListener listener = new AddInterestListener();
47       button.addActionListener(listener);
48
49       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
50       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51       frame.setVisible(true);
52    }
53
54    private static final double INTEREST_RATE = 10;
55    private static final double INITIAL_BALANCE = 1000;
56
57    private static final int FRAME_WIDTH = 400;
58    private static final int FRAME_HEIGHT = 100;
59 }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

17. Como você coloca a mensagem `"balance: . . ."` à esquerda do botão `"AddInterest"`?
18. Por que não foi necessário declarar a variável `button` como `final`?

ERRO COMUM 9.4

Esquecendo de relacionar um ouvinte

Se você executar seu programa e descobrir que seus botões parecem não funcionar, verifique novamente se você relacionou o ouvinte de botão. O mesmo é válido para os outros componentes da interface com o usuário. É um erro bastante comum programar a classe ouvinte e a ação da rotina de tratamento de evento sem realmente relacionar o ouvinte à fonte do evento.

DICA DE PRODUTIVIDADE 9.1

Não utilize um contêiner como um ouvinte

Neste livro, utilizamos classes internas para os ouvintes de evento. Essa abordagem funciona para vários tipos de evento. Depois de dominar a técnica, você não precisará pensar mais sobre isso. Muitos ambientes de desenvolvimento geram automaticamente código com classes internas, portanto, é uma boa idéia entendê-los.

Mas alguns programadores pulam as classes ouvintes de evento e, em vez disso, transformam um contêiner (por exemplo, um painel ou um frame) em um ouvinte. Eis um exemplo típico. O método `actionPerformed` é adicionado à classe do visualizador. Isto é, o visualizador implementa a interface `ActionListener`.

```
public class InvestmentViewer
      implements ActionListener // Essa abordagem não é recomendável
{
   public InvestmentViewer()
   {
      JButton button = new JButton("Add Interest");
      button.addActionListener(this);
      . . .
   }
   public void actionPerformed(ActionEvent event)
   {
   }
   . . .
}
```

Agora o método `actionPerformed` é uma parte da classe `InvestmentViewer` em vez de uma parte de uma classe ouvinte separada. O ouvinte é instalado como `this`.

Essa técnica tem duas falhas importantes. Primeiro, ela separa a definição do botão, da ação do botão. Além disso, ela não oferece um bom *escalonamento*. Se a classe do visualizador contiver dois botões que geram eventos de ação, o método `actionPerformed` deve então descobrir a fonte do evento, o que resulta em um código cansativo e propenso a erros.

As Seções 9.9 e 9.10, disponíveis no material online do livro no site www.bookman.com.br, mostram como você pode processar eventos de temporizadores e de mouse.

Erro comum 9.5

Esquecendo de repintar

Tenha cuidado quando suas rotinas de tratamento de evento alterarem os dados em um componente pintado. Quando você faz uma alteração nos dados, o componente não é automaticamente pintado de acordo com os novos dados. Você deve instruir o framework Swing de que o componente precisa ser repintado, chamando o método `repaint` na rotina de tratamento de evento ou nos métodos modificadores do componente. O método `paintComponent` do seu componente será então invocado, em um momento oportuno, com um objeto `Graphics` apropriado. Observe que você não deve chamar o método `paintComponent` diretamente.

Essa é uma preocupação apenas dos seus próprios componentes. Quando você faz uma alteração em um componente Swing padrão, como um `JLabel`, o componente é automaticamente repintado.

Tópico avançado 9.3

Adaptadores de evento

O Tópico Avançado 9.3 mostra como utilizar classes adaptadoras de evento para simplificar as classes ouvintes.

Fato aleatório 9.2

Linguagens de programação

O Fato Aleatório 9.2 investiga a história de várias linguagens de programação comuns.

RESUMO DO CAPÍTULO

1. Utilize tipos interface para tornar o código mais reutilizável.
2. Um tipo interface Java declara um conjunto de métodos e suas assinaturas.
3. Diferentemente de uma classe, um tipo interface não fornece nenhuma implementação.
4. Utilize a palavra-chave `implements` para indicar que uma classe implementa um tipo interface.
5. Interfaces podem reduzir o acoplamento entre classes.
6. Você pode converter de um tipo classe para um tipo interface, desde que a classe implemente a interface.
7. Você precisa fazer uma coerção, ou typecasting, para converter entre um tipo interface e um tipo classe.
8. O polimorfismo indica o princípio de que o comportamento pode variar com base no tipo real de um objeto.
9. A vinculação inicial dos métodos ocorre se o compilador selecionar um método entre vários possíveis candidatos. A vinculação tardia ocorre se a seleção do método acontecer quando o programa é executado.
10. Uma classe interna é declarada dentro de outra classe. Classes internas são comumente utilizadas como classes táticas, que não devem ser visíveis em outra parte de um programa.
11. Eventos da interface com o usuário incluem o pressionamento de uma tecla, movimentos do mouse, cliques em botões, seleções de menu e assim por diante.
12. Um ouvinte de evento pertence a uma classe que é fornecida pelo programador da aplicação. Os métodos desse ouvinte descrevem as ações a serem tomadas quando um evento ocorre.
13. Fontes de evento informam sobre os eventos. Quando um evento ocorre, a fonte do evento notifica todos os ouvintes do evento.
14. Use os componentes `JButton` para botões. Anexe um `ActionListener` a cada botão.
15. Métodos de uma classe interna podem acessar variáveis locais de blocos adjacentes e campos que cercam as classes.
16. Variáveis locais que são acessadas por um método de classe interna devem ser declaradas como `final`.
17. Utilize um contêiner `JPanel` para agrupar múltiplos componentes da interface com o usuário.
18. Freqüentemente você instala ouvintes de evento como classes internas para que possam ter acesso aos campos adjacentes, métodos e variáveis finais.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.awt.Component
   addMouseListener
   repaint
java.awt.Container
   add
java.awt.Rectangle
   setLocation
java.awt.event.ActionListener
   actionPerformed
java.awt.event.MouseEvent
   getX
   getY
java.awt.event.MouseListener
   mouseClicked
   mouseEntered
   mouseExited
   mousePressed
   mouseReleased
javax.swing.AbstractButton
   addActionListener
javax.swing.JButton
javax.swing.JLabel
javax.swing.JPanel
javax.swing.Timer
   start
   stop
```

EXERCÍCIOS DE REVISÃO

★ **Exercício R9.1.** Suponha que `C` seja uma classe que implementa as interfaces `I` e `J`. Quais atribuições a seguir requerem uma coerção?

```
C c = . . .;
I i = . . .;
J j = . . .;
```

a. `c = i;`

b. `j = c;`

c. `i = j;`

★ **Exercício R9.2.** Suponha que `C` seja uma classe que implementa as interfaces `I` e `J` e suponha que `i` seja declarado como:

```
I i = new C();
```

Quais instruções a seguir lançarão uma exceção?

a. `C c = (C) i;`

b. `J j = (J) i;`

c. `i = (I) null;`

★ **Exercício R9.3.** Suponha que a classe `Sandwich` implemente a interface `Edible` e você conhece as definições das variáveis

```
Sandwich sub = new Sandwich();
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
Edible e = null;
```

Quais instruções de atribuição a seguir são válidas?

a. `e = sub;`

b. `sub = e;`

c. `sub = (Sandwich) e;`

d. `sub = (Sandwich) cerealBox;`

e. `e = cerealBox;`

f. `e = (Edible) cerealBox;`

g. `e = (Rectangle) cerealBox;`

h. `e = (Rectangle) null;`

★★ **Exercício R9.4.** Como uma coerção como `(BankAccount) x` difere de uma coerção dos valores de número como `(int) x`?

★★ **Exercício R9.5.** As classes `Rectangle2D.Double`, `Ellipse2D.Double` e `Line2D.Double` implementam a interface `Shape`. A classe `Graphics2D` depende da interface `Shape`, mas não das classes retângulo, elipse e linha. Desenhe um diagrama UML para indicar esses fatos.

★★ **Exercício R9.6.** Suponha que `r` contenha uma referência a um `new Rectangle(5, 10, 20, 30)`. Qual das atribuições a seguir é válida? (Examine a documentação da API para verificar quais interfaces a classe `Rectangle` implementa.)

a. `Rectangle a = r;`

b. `Shape b = r;`

c. `String c = r;`

d. `ActionListener d = r;`

e. `Measurable e = r;`

f. `Serializable f = r;`

g. `Object g = r;`

★★ **Exercício R9.7.** Classes como `Rectangle2D.Double`, `Ellipse2D.Double` e `Line2D.Double` implementam a interface `Shape`. A interface `Shape` tem um método

```
Rectangle getBounds()
```

que retorna um retângulo incluindo completamente a forma geométrica. Considere a chamada de método:

```
Shape s = . . .;
Rectangle r = s.getBounds();
```

Explique por que isso é um exemplo de polimorfismo.

★★★ **Exercício R9.8.** Em Java, uma chamada de método como `x.f()` usa a vinculação tardia – o método exato a ser chamado depende do tipo do objeto que `x` referencia. Forneça dois tipos de chamadas de método que utilizam a vinculação inicial em Java.

★★ **Exercício R9.9.** Suponha que você precise processar um array de funcionários para localizar o salário médio e o salário mais alto. Discuta o que você precisa fazer para utilizar a implementação da classe `DataSet` na Seção 9.1 (que processa objetos `Measurable`). O que você precisa fazer para utilizar a segunda implementação (na Seção 9.4)? Qual é mais fácil?

★★★ **Exercício R9.10.** O que acontece se você adicionar um objeto `String` à implementação da classe `DataSet` na Seção 9.1? O que acontece se você adicionar um objeto `String` a um objeto `DataSet` da implementação na Seção 9.4 que usa uma classe `RectangleMeasurer`?

★ **Exercício R9.11.** Como você reorganizaria o programa `DataSetTester3` se precisasse tornar `RectangleMeasurer` uma classe de primeiro nível (isto é, não uma classe interna)?

★★ **Exercício R9.12.** O que é um retorno de chamada? Você pode pensar em um outro uso para um retorno de chamada para a classe `DataSet`? (*Dica:* Exercício P9.8.)

★★ **Exercício R9.13.** Considere esta classe interna e de primeiro nível. Quais variáveis o método `f` pode acessar?

```
public class T
{
   public void m(final int x, int y)
   {
      int a;
      final int b;
      class C implements I
      {
         public void f()
         {
            . . .
         }
      }
      final int c;
      . . .
   }
   private int t;
}
```

★★ **Exercício R9.14.** O que acontece quando uma classe interna tenta acessar uma variável local não-`final`? Teste e explique seus resultados.

★★★G **Exercício R9.15.** Como você reorganizaria o programa `InvestmentViewer1` se precisasse tornar `AddInterestListener` uma classe de primeiro nível (isto é, não uma classe interna)?

★G **Exercício R9.16.** O que é um objeto evento? Uma fonte de evento? Um ouvinte de evento?

★G **Exercício R9.17.** Da perspectiva de um programador, qual é a diferença mais importante entre as interfaces com o usuário de uma aplicação de console e de uma aplicação gráfica?

★G **Exercício R9.18.** Qual é a diferença entre um `ActionEvent` e um `MouseEvent`?

★★G **Exercício R9.19.** Por que a interface `ActionListener` tem apenas um método, enquanto a `MouseListener` tem cinco métodos?

★★G **Exercício R9.20.** Uma classe pode ser uma fonte de evento para vários tipos de evento? Se sim, dê um exemplo.

★★G **Exercício R9.21.** Quais informações um objeto evento de ação transporta? Que informações adicionais um objeto evento de mouse carrega?

★★★G **Exercício R9.22.** Por que estamos utilizando classes internas para ouvintes de evento? Se Java não tivesse classes internas, ainda seria possível implementar ouvintes de evento? Como?

★★G **Exercício R9.23.** Qual é a diferença entre os métodos `paintComponent` e `repaint`?

★G **Exercício R9.24.** Qual é a diferença entre um frame e um painel?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P9.1.** Faça a classe `Die` do Capítulo 6 implementar a interface `Measurable`. Gere dados, aplique uma coerção e adicione-os à implementação da classe `DataSet` na Seção 9.1. Exiba a média.

★ **Exercício P9.2.** Defina uma classe `Quiz` que implementa a interface `Measurable`. Um questionário tem uma pontuação e uma nota baseada em letras (como `B+`). Utilize a implementação da classe `DataSet` na Seção 9.1 para processar uma coleção de questionários. Exiba a pontuação média e o questionário com a pontuação mais alta (com uma nota baseada em letra e uma pontuação).

★ **Exercício P9.3.** Uma pessoa tem um nome e uma altura em centímetros. Utilize a implementação da classe `DataSet` na Seção 9.4 para processar uma coleção de objetos `Person`. Exiba a altura média e o nome da pessoa mais alta.

★ **Exercício P9.4.** Modifique a implementação da classe `DataSet` na Seção 9.1 (aquela que processa objetos `Measurable`) para que ela também calcule o elemento mínimo.

★ **Exercício P9.5.** Modifique a implementação da classe `DataSet` na Seção 9.4 (aquela que usa um objeto `Measurer`) para que ela também calcule o elemento mínimo.

★ **Exercício P9.6.** Utilizando um objeto `Measurer` diferente, processe um conjunto de objetos `Rectangle` para localizar o retângulo com o maior perímetro.

★★★ **Exercício P9.7.** Aprimore a classe `DataSet` para que ela possa ser utilizada com um objeto `Measurer` ou para processar objetos `Measurable`. *Dica:* Forneça um construtor padrão que implementa um `Measurer` que processa objetos `Measurable`.

★★ **Exercício P9.8.** Defina uma interface `Filter` desta maneira:

```
public interface Filter
{
   boolean accept(Object x);
}
```

Modifique a implementação da classe `DataSet` na Seção 9.4 para que ela utilize tanto um `Measurer` como um objeto `Filter`. Somente os objetos que o filtro aceita devem ser processados. Demonstre sua modificação fazendo com que um conjunto de dados processe uma coleção de contas bancárias, filtrando todas as contas com saldo menor que US$ 1.000.

★★ **Exercício P9.9.** Pesquise a definição da interface `Comparable` padrão na documentação da API. Modifique a classe `DataSet` da Seção 9.1 para que ela aceite objetos `Comparable`. Com essa interface, não é mais significativo calcular a média. A classe `DataSet` deve registrar os valores mínimos e máximos dos dados. Teste sua classe `DataSet` modificada adicionando alguns objetos `String`. (A classe `String` implementa a interface `Comparable`.)

★ **Exercício P9.10.** Modifique a classe `Coin` para que ela implemente a interface `Comparable`.

★★★ **Exercício P9.11.** O método `System.out.printf` predefiniu formatos para impressão de inteiros, de números de ponto flutuante e de outros tipos de dados. Mas ele também é extensível. Se utilizar o formato `S`, você pode imprimir qualquer classe que implementa a interface `Formattable`. Essa interface tem um único método:

```
void formatTo(Formatter formatter, int flags, int width, int precision)
```

Neste exercício, você deve fazer a classe `BankAccount` implementar a interface `Formattable`. Ignore os flags e formate o saldo bancário de uma maneira precisa e simples, utilizando a largura dada. Para executar essa tarefa, você precisa obter uma referência `Appendable` como esta:

```
Appendable a = formatter.out();
```

`Appendable` é outra interface com um método:

```
void append(CharSequence sequence)
```

`CharSequence` é uma outra interface implementada pela classe `String` (entre outras). Construa uma string primeiro convertendo o saldo bancário em uma string e então a preenchendo com espaços para que tenha a largura desejada. Passe essa string para o método `append`.

★★★ **Exercício P9.12.** Aprimore o método `formatTo` do Exercício P9.11 levando em consideração a precisão.

★★G **Exercício P9.13.** Escreva um método `randomShape` que gera aleatoriamente objetos que implementam a interface `Shape`: alguma combinação de retângulos, elipses e linhas com posições aleatórias. Chame-o 10 vezes e desenhe todos eles.

★G **Exercício P9.14.** Aprimore o programa `ButtonViewer` para que ele imprima uma mensagem "I was clicked *n* times!" [Fui clicado *n* vezes!] sempre que o botão é clicado. O valor *n* deve ser incrementado a cada clique.

★★G **Exercício P9.15.** Aprimore o programa `ButtonViewer` para que tenha dois botões. Cada um deles imprime uma mensagem "I was clicked *n* times!" sempre que o botão é clicado. Cada botão deve ter uma contagem de cliques separada.

★★G **Exercício P9.16.** Aprimore o programa `ButtonViewer` para que ele tenha dois botões A e B rotulados. Cada um deles imprime uma mensagem "Button *x* was clicked!", onde *x* é A ou B.

★★★G **Exercício P9.17.** Implemente um programa `ButtonViewer` como no Exercício P9.16, utilizando apenas uma única classe ouvinte.

★G **Exercício P9.18.** Aprimore o programa `ButtonViewer` para que ele imprima o momento em que o botão foi clicado.

★★★G **Exercício P9.19.** Implemente o `AddInterestListener` no programa `InvestmentViewer1` como uma classe normal (isto é, não como uma classe interna). *Dica:* armazene uma referência à conta bancária. Adicione um construtor à classe ouvinte que configura a referência.

★★★G **Exercício P9.20.** Implemente o `AddInterestListener` no programa `InvestmentViewer2` como uma classe normal (isto é, não como uma classe interna). *Dica:* armazene referências à conta bancária e ao rótulo no ouvinte. Adicione um construtor à classe ouvinte que configura as referências.

★★G **Exercício P9.21.** Escreva um programa que usa um timer para imprimir a hora atual a cada um segundo. *Dica:* o código a seguir imprime a data atual:

```
Date now = new Date();
System.out.println(now);
```

A classe `Date` está no pacote `java.util`.

★★★G **Exercício P9.22.** Altere `RectangleComponent` para o programa de animação na Seção 9.9 a fim de que o retângulo rebata contra as bordas do componente em vez de simplesmente mover-se para fora dele.

★★G **Exercício P9.23.** Escreva um programa que cria a animação de um carro e faça ele se mover de um lado a outro em um frame.

★★★G **Exercício P9.24.** Escreva um programa que cria a animação de dois carros que se movem de um lado a outro no frame em direções opostas (mas em alturas diferentes para que não colidam).

★★G **Exercício P9.25.** Altere `RectangleComponent` para o programa ouvinte de mouse na Seção 9.9 de modo que um novo retângulo seja adicionado ao componente sempre que clicarmos com o mouse. Dica: mantenha um `ArrayList<Rectangle>` e desenhe todos os retângulos no método `paintComponent`.

★★G **Exercício P9.26.** Escreva um programa que demonstra o crescimento de uma população de baratas. Inicie com duas baratas e dobre o número de baratas com cada clique de botão.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 9.1.** Projete uma interface `MoveableShape` que pode ser utilizada como um mecanismo genérico para criar a animação de uma forma geométrica. Uma forma móvel deve ter dois métodos: `move` e `draw`. Escreva um `AnimationPanel` genérico que pinte e mova qualquer `MoveableShape` (ou lista de arrays de objetos `MoveableShape` se você leu o Capítulo 7). Forneça a forma de um retângulo móvel e do carro.

★★★ **Projeto 9.2.** Sua tarefa é projetar um programa genérico para gerenciar jogos de tabuleiro com dois jogadores. Seu programa deve ser suficientemente flexível para tratar jogos como jogo-da-velha, xadrez ou o jogo Nim do Projeto 6.2.

Projete uma interface `Game` que descreve um jogo de tabuleiro. Pense sobre o que seu programa precisa fazer. Ele solicita que o primeiro jogador insira um movimento – uma string em um formato específico de jogo, como `Be3` no xadrez. Seu programa não conhece nada sobre jogos específicos, portanto a interface `Game` deve ter um método como

```
boolean isValidMove(String move)
```

Quando a jogada é considerada válida, ela precisa ser executada – a interface precisa de outro método, `executeMove`. Em seguida, seu programa precisa verificar se o jogo terminou. Se não, o movimento do outro jogador é processado. Você também deve fornecer algum mecanismo para exibir o estado atual do tabuleiro.

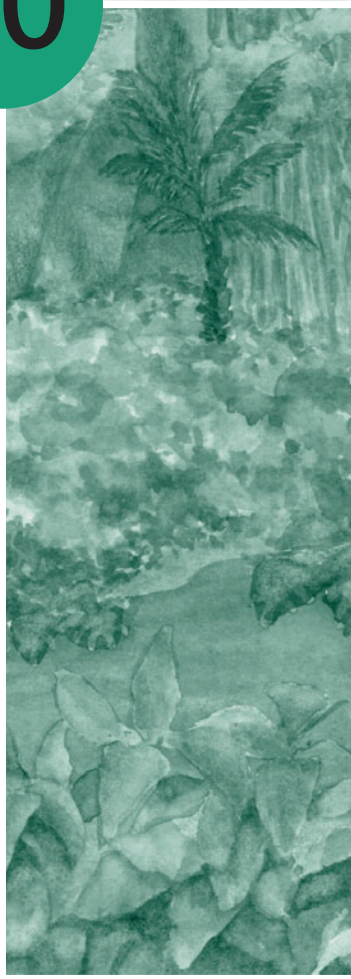
Projete a interface `Game` e forneça duas implementações que preferir – como `Nim` e `Chess` (ou `TicTacToe` se você for menos ambicioso). Sua classe `GamePlayer` deve gerenciar uma referência `Game` sem saber qual jogo é reproduzido e processar o movimento dos dois jogadores. Forneça dois programas que só diferem na inicialização da referência `Game`.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Você deve implementar a interface `Measurable` e seu método `getMeasure` deve retornar a população.
2. A classe `Object` não tem um método `getMeasure` e o método `add` invoca o método `getMeasure`.
3. Somente se `x` realmente referenciar um objeto `BankAccount`.
4. Não – uma referência `Coin` pode ser convertida em uma referência `Measurable`, mas se você tentar fazer uma coerção nessa referência para um `BankAccount`, ocorrerá uma exceção.
5. `Measurable` é uma interface. Interfaces não têm nenhuma implementação de método e de campo.
6. Essa variável nunca referencia um objeto `Measurable`. Ela referencia um objeto de alguma classe – uma classe que implemente a interface `Measurable`.
7. Os dois descrevem uma situação em que um nome de método pode indicar múltiplos métodos. Mas a sobrecarga é resolvida no início pelo compilador, examinando os tipos das variáveis de parâmetro. O polimorfismo é resolvido tardiamente, examinando o tipo do objeto de parâmetro implícito um pouco antes de fazer a chamada.
8. A classe `String` não implementa a interface `Measurable`.
9. Implemente uma classe `StringMeasurer` que implementa a interface `Measurer`.
10. Um medidor mede um objeto, enquanto `getMeasure` mede a "si próprio", isto é, o parâmetro implícito.
11. Classes internas são convenientes para classes insignificantes. Além disso, os métodos dessas classes podem acessar variáveis e campos a partir do escopo adjacente.
12. Quatro: um para a classe externa, um para a classe interna e dois para as classes `DataSet` e `Measurer`.
13. O objeto `button` é a fonte de eventos. O objeto `listener` é o ouvinte de eventos.
14. A classe `ClickListener` implementa a interface `ActionListener`.
15. O acesso direto é mais simples que a alternativa – passar a variável como um parâmetro para um construtor ou para um método.
16. A variável local deve ser declarada como `final`.
17. Primeiro adicione `label` ao `panel`, depois adicione `button`.
18. O método `actionPerformed` não acessa essa variável.

Capítulo 10

Herança



Objetivos do capítulo

- Aprender sobre herança
- Entender como herdar e sobrescrever métodos de superclasses
- Ser capaz de invocar construtores de superclasses
- Aprender sobre controle de acesso protegido e controle de acesso de pacote
- Entender a superclasse comum `Object` e como sobrescrever seus métodos `toString` e `equals`

G Utilizar herança para personalizar interfaces com o usuário

Neste capítulo, discutiremos o importante conceito de herança. É possível criar classes especializadas que herdam o comportamento de classes mais gerais. Você aprenderá a implementar herança em Java e como utilizar a classe `Object` – a classe mais geral na hierarquia de herança.

Conteúdo do Capítulo

10.1 Uma introdução à herança 400
Sintaxe 10.1: Herança 403
Erro comum 10.1: Confundindo superclasses e subclasses 404

10.2 Hierarquias de heranças 405

10.3 Herdando campos e métodos de instância 407
Sintaxe 10.2: Chamando um método de superclasse 411
Erro comum 10.2: Sombreando campos de instância 411
Erro comum 10.3: Falhando ao invocar o método da superclasse 412

10.4 Construção das subclasses 413
Sintaxe 10.3: Chamando um construtor da superclasse 413

10.5 Convertendo subclasses em superclasses 414
Sintaxe 10.4: O operador `instanceof` 416

10.6 Polimorfismo 417
Tópico avançado 10.1: Classes abstratas ⊕
Tópico avançado 10.2: Métodos e classes final ⊕

10.7 Controle de acesso 422
Erro comum 10.4: Acesso acidental a pacote 424
Erro comum 10.5: Tornando métodos herdados menos acessíveis 424
Tópico avançado 10.3: Acesso protegido ⊕

10.8 Object: a superclasse cósmica 425
Dica de produtividade 10.1: Fornecer `toString` em todas as classes 427
Tópico avançado 10.4: Herança e o método `toString` ⊕
Erro comum 10.6: Definindo o método `equals` com o tipo errado de parâmetro 429
Tópico avançado 10.5: Herança e o método `equals` ⊕
Erro comum 10.7: Esquecendo de clonar 430
Dica de qualidade 10.1: Clone campos de instância mutáveis em métodos de acesso ⊕
Tópico avançado 10.6: Implementando o método `clone` ⊕
Tópico avançado 10.7: Tipos enumerados revistos ⊕
Fato aleatório 10.1: Linguagens de criação de scripts ⊕

10.9G Utilizando herança para personalizar frames 432
Tópico avançado 10.8: Adicionando o método `main` à classe de frame 433

10.10G Processando entrada de texto 434

10.11G Áreas de texto 437
Como fazer 10.1: Implementando uma interface gráfica com o usuário (*graphical user interface* – GUI) 439
Erro comum 10.8: Por padrão, componentes têm largura e altura zero 441
Dica de produtividade 10.2: Reutilização de código 441

10.1 Uma introdução à herança

A herança é um mecanismo para estender as classes existentes adicionando novos métodos e campos.

A herança é um mecanismo para aprimorar as classes existentes. Se precisar implementar uma nova classe e uma classe que representa um conceito mais geral já estiver disponível, então a nova classe pode herdar da classe existente. Por exemplo, suponha que você precise definir uma classe `SavingsAccount` para modelar uma conta que paga uma taxa fixa de juros sobre os depósitos. Você já tem uma classe `BankAccount` e uma conta-poupança é um caso especial de uma conta bancária. Nessa situação, faz sentido utilizar a construção de herança da linguagem. Eis a sintaxe para a definição da classe:

```
class SavingsAccount extends BankAccount
{
   novos métodos
   novos campos de instância
}
```

Na definição da classe `SavingsAccount` você só especifica os novos métodos e campos de instância. A classe `SavingsAccount` *herda automaticamente* todos os métodos e campos de instância da classe `BankAccount`. Por exemplo, o método `deposit` é aplicado automaticamente às contas de poupança:

```
SavingsAccount collegeFund = new SavingsAccount(10);
   // Conta-poupança com 10% de juros
collegeFund.deposit(500);
   // OK usar o método de BankAccount com o objeto SavingsAccount
```

A classe mais geral chama-se superclasse. A classe mais especializada que herda da superclasse chama-se subclasse.

Toda classe estende a classe `Object` direta ou indiretamente.

Devemos introduzir uma nova terminologia aqui. A classe mais geral que forma a base para a herança é chamada de superclasse. A classe mais especializada, que herda da superclasse, chama-se subclasse. No nosso exemplo, `BankAccount` é a superclasse e `SavingsAccount` é a subclasse.

Em Java, toda classe que não estende especificamente uma outra é uma subclasse da classe `Object`. Por exemplo, a classe `BankAccount` estende a classe `Object`. A classe `Object` tem alguns métodos que fazem sentido para todos os objetos, como o método `toString`, que você pode utilizar para obter uma string que descreve o estado de um objeto.

A Figura 1 apresenta um diagrama de classes que mostra o relacionamento entre as três classes `Object`, `BankAccount` e `SavingsAccount`. Em um diagrama de classes, você indica a herança através de uma seta sólida, com uma ponta em forma de "triângulo vazado" que aponta para a superclasse.

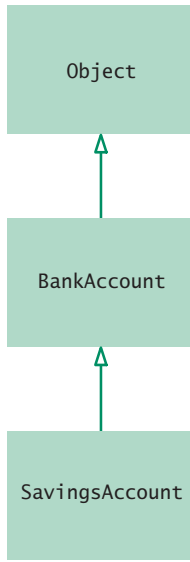


Figura 1
Um diagrama de herança.

A herança a partir de uma classe é diferente da implementação de uma interface: a subclasse herda o comportamento e o estado da superclasse.

Nesse ponto você poderia perguntar em que a herança difere da implementação de uma interface. Uma interface não é uma classe. Ela não tem *um estado ou um comportamento*. Ela apenas informa quais métodos você deve implementar. Uma superclasse tem um estado e um comportamento e as subclasses herdam esse estado e comportamento.

Uma vantagem da herança é a reutilização de código.

Uma razão importante para a herança é *a reutilização de código*. Herdando uma classe existente, você não tem de replicar o esforço necessário para projetar e aperfeiçoar essa classe. Por exemplo, ao implementar a classe `SavingsAccount`, você pode contar com os métodos `withdraw`, `deposit` e `getBalance` da classe `BankAccount` sem tocá-los.

Ao definir uma subclasse, você especifica campos de instância adicionais, métodos adicionais e métodos modificados ou redefinidos.

Vejamos que objetos da classe conta-poupança são diferentes de objetos da classe `BankAccount`. Iremos configurar um taxa de juros no construtor e precisaremos de um método para aplicar esses juros periodicamente. Isto é, além dos três métodos que podem ser aplicados a cada conta, há um método adicional `addInterest`. O novo método e campo de instância devem ser definidos na subclasse.

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      Implementação do construtor
   }
   public void addInterest()
   {
      Implementação do método
   }
   private double interestRate;
}
```

A Figura 2 mostra o leiaute de um objeto `SavingsAccount`. Ele herda o campo de instância `balance` da superclasse `BankAccount` e ganha um campo de instância adicional: `interestRate`.

Em seguida, você precisa implementar o novo método `addInterest`. O método calcula os juros devidos no saldo atual e acrescenta esses juros à conta.

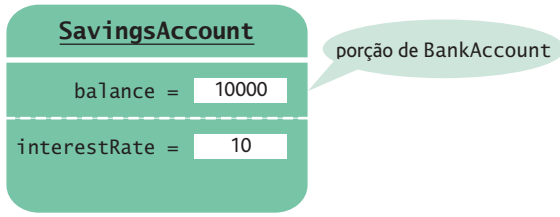


Figura 2 Leiaute de um objeto de subclasse.

SINTAXE 10.1 Herança

```
class NomeDaSubClasse extends NomeDaSuperClasse
{
   métodos
   campos de instância
}
```

Exemplo:

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }
   private double interestRate;
}
```

Objetivo:

Definir uma nova classe que herda de uma classe existente e definir os métodos e campos de instância que são adicionados à nova classe

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }
   private double interestRate;
}
```

Você pode perguntar por que o método `addInterest` chama os métodos `getBalance` e `deposit` em vez de atualizar diretamente o campo `balance` da superclasse. Essa é uma conseqüência do encapsulamento. O campo `balance` foi definido como `private` na classe `BankAccount`. O método `addInterest` é definido na classe `SavingsAccount`. Ele não tem permissão de acessar um campo privado de uma outra classe.

Observe como o método `addInterest` chama os métodos `getBalance` e `deposit` da superclasse sem especificar um parâmetro implícito. Isso significa que as chamadas são aplicadas ao mesmo objeto, isto é, o parâmetro implícito do método `addInterest`.

Por exemplo, se você chamar

```
collegeFund.addInterest();
```

então as instruções a seguir são executadas:

```
double interest = collegeFund.getBalance()
      * collegeFund.interestRate / 100;
collegeFund.deposit(interest);
```

Em outras palavras, as instruções no método `addInterest` são uma abreviação para as seguintes instruções:

```
double interest = this.getBalance()
      * this.interestRate / 100;
this.deposit(interest);
```

(Lembre-se de que a variável `this` contém uma referência ao parâmetro implícito.)

Autoverificação da aprendizagem

1. Quais campos de instância um objeto da classe `SavingsAccount` têm?
2. Nomeie quatro métodos que você pode aplicar a objetos `SavingsAccount`.
3. Se a classe `Manager` estende a classe `Employee`, qual é a superclasse e qual é a subclasse?

Erro comum 10.1

Confundindo superclasses e subclasses

Se comparar um objeto do tipo `SavingsAccount` com um objeto do tipo `BankAccount`, você então descobrirá que:

- A palavra-chave `extends` sugere que o objeto `SavingsAccount` é uma versão estendida de um `BankAccount`.
- O objeto `SavingsAccount` é maior; ele tem um campo de instância `interestRate` extra.
- O objeto `SavingsAccount` é mais capaz; ele tem um método `addInterest`.

Sob todos os aspectos, ele parece um objeto superior. Portanto, por que `SavingsAccount` chama-se *subclasse* e `BankAccount` *superclasse*?

A terminologia *super/sub* se origina da teoria dos conjuntos. Examine o conjunto de todas as contas bancárias. Nem todas são objetos `SavingsAccount`; algumas delas são outros tipos de contas bancárias. Portanto, o conjunto de objetos `SavingsAccount` é um *subconjunto* do conjunto de todos os objetos `BankAccount` e o conjunto de objetos `BankAccount` é um *superconjunto* do conjunto de objetos `SavingsAccount`. Os objetos mais especializados no subconjunto têm um estado mais rico e mais capacidades.

10.2 Hierarquias de heranças

No mundo real, freqüentemente você categoriza conceitos em *hierarquias.* Hierarquias costumam ser representadas como árvores, com os conceitos mais gerais na raiz da hierarquia e os mais especializados nas ramificações. A Figura 3 mostra um exemplo típico.

Conjuntos de classes podem formar hierarquias complexas de herança.

Em Java, é igualmente comum agrupar classes em *hierarquias de herança* complexas. As classes que representam os conceitos mais gerais estão próximas à raiz, classes mais especializadas estão próximas dos ramos. Por exemplo, a Figura 4 mostra parte da hierarquia dos componentes Swing de interface com o usuário em Java.

Ao projetar uma hierarquia de classes, você pensa nos recursos e comportamentos comuns a todas as classes que você está projetando. Essas propriedades comuns são agrupadas em uma superclasse. Por exemplo, todos os componentes da interface com o usuário têm uma largura e uma altura e os métodos `getWidth` e `getHeight` da classe `JComponent` retornam as dimensões dos componentes. Propriedades mais especializadas podem ser encontradas nas subclasses. Por exemplo, botões podem ter texto e rótulos de ícone. A classe `AbstractButton`, mas não a superclasse `JComponent`, contém métodos para configurar e obter o texto do botão, o ícone e os campos de instância para armazená-los. As classes botão individuais (como `JButton`, `JRadioButton` e `JCheckBox`) herdam essas propriedades. Na realidade, a classe `AbstractButton` foi criada para expressar a semelhança entre esses botões.

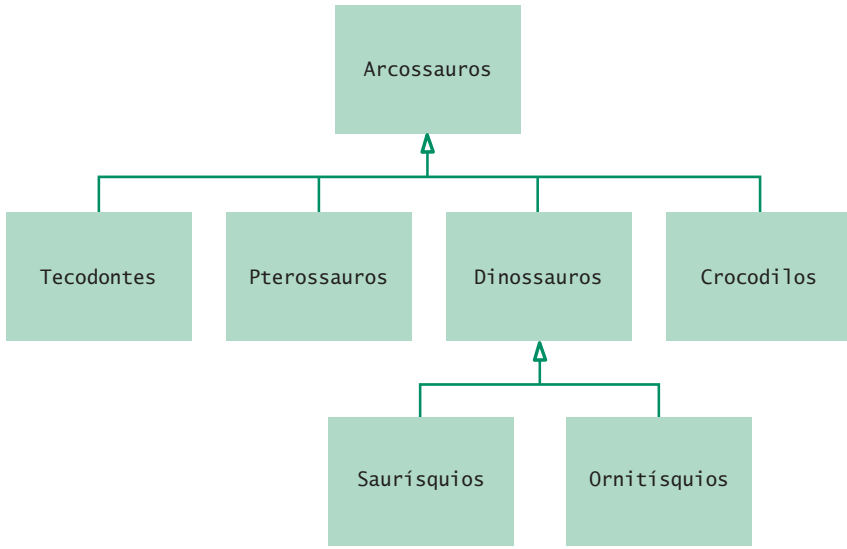


Figura 3 Parte da hierarquia dos répteis antigos.

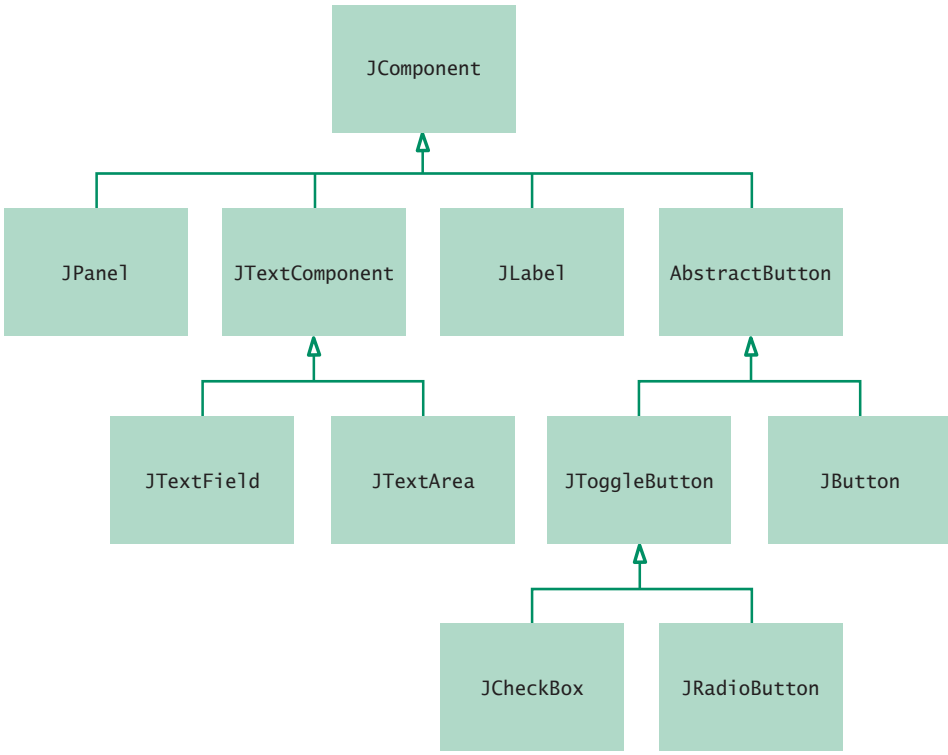


Figura 4 Parte da hierarquia dos componentes Swing de interface com o usuário.

Utilizaremos um exemplo mais simples de hierarquia no nosso estudo dos conceitos de herança. Considere um banco que oferece aos seus clientes os seguintes tipos de conta:

1. A conta corrente não rende juros, ela oferece algumas transações gratuitas mensalmente e cobra uma tarifa por cada transação adicional.
2. A conta poupança rende juros capitalizados mensalmente. (Na nossa implementação, os juros são capitalizados utilizando o saldo do último dia do mês, o que não é realista. Em geral, os bancos utilizam um saldo médio ou o saldo diário mínimo. O Exercício P10.1 solicita que você implemente essa melhoria.)

A Figura 5 mostra a hierarquia de herança. O Exercício P10.2 solicita que você adicione uma outra classe a essa hierarquia.

Em seguida, vamos determinar o comportamento dessas classes. Todas as contas bancárias suportam o método `getBalance`, que simplesmente informa o saldo atual. Elas também suportam os métodos `deposit` e `withdraw`, embora os detalhes da implementação sejam diferentes. Por exemplo, uma conta corrente deve monitorar o número de transações para poder cobrar as tarifas por transação.

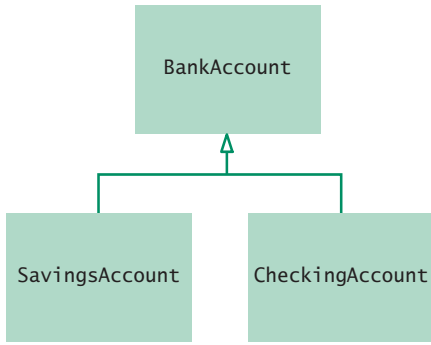


Figura 5 Hierarquia de herança das classes de contas bancárias.

A conta corrente precisa de um método `deductFees` para subtrair as tarifas mensais e redefinir o contador de transações. Os métodos `deposit` e `withdraw` devem ser redefinidos para contar as transações.

A conta poupança precisa de um método `addInterest` para adicionar juros.

Resumindo: as subclasses suportam todos os métodos da superclasse, mas suas implementações podem ser modificadas para que correspondam aos objetivos especializados das subclasses. Além disso, as subclasses são livres para introduzir métodos adicionais.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

4. Qual é o propósito da classe `JTextComponent` na Figura 4?
5. Qual campo de instância precisaremos adicionar à classe `CheckingAccount`?

10.3 Herdando campos e métodos de instância

Ao formar uma subclasse a partir de uma dada classe, você pode especificar métodos e campos de instância adicionais. Nesta seção discutiremos esse processo em detalhes.

Ao definir os métodos para uma subclasse, há três possibilidades.

1. Você pode sobrescrever *métodos da superclasse*. Se especificar um método com a mesma *assinatura* (isto é, o mesmo nome e os mesmos tipos de parâmetro), ele sobrescreve o método com o mesmo nome na superclasse. Sempre que o método é aplicado a um objeto do tipo da subclasse, o método que sobrescreve, e não o método original, é executado. Por exemplo, `CheckingAccount.deposit` sobrescreve `BankAccount.deposit`.
2. Você pode *herdar* métodos da superclasse. Se não sobrescrever explicitamente um método da superclasse, você irá herdá-lo automaticamente. O método da superclasse pode ser aplicado aos objetos da subclasse. Por exemplo, a classe `SavingsAccount` herda o método `BankAccount.getBalance`.

3. Você pode definir novos métodos. Se definir um método que não existia na superclasse, o novo método só pode ser aplicado a objetos da subclasse. Por exemplo, `SavingsAccount.addInterest` é um novo método que não existe na superclasse `BankAccount`.

A situação para campos de instância é bem diferente. Você não pode sobrescrever campos de instância. Para campos em uma subclasse, há somente dois casos:

1. A subclasse herda todos os campos da superclasse. Todos os campos de instância da superclasse são herdados automaticamente. Por exemplo, todas as subclasses da classe `BankAccount` herdam o campo de instância `balance`.
2. Qualquer novo campo de instância que você definir na subclasse só estará presente nos objetos da subclasse. Por exemplo, a subclasse `SavingsAccount` define um novo campo de instância `interestRate`.

O que acontece se você definir um novo campo com o mesmo nome de um campo da superclasse? Por exemplo, você pode definir outro nome de campo `balance` na classe `SavingsAccount`? Isso é possível, mas extremamente indesejável. Cada objeto `SavingsAccount` teria *dois* campos de instância com o mesmo nome. Esses dois campos podem armazenar valores diferentes, o que provavelmente resulta em confusões – veja Erro comum 10.2.

Já implementamos as classes `BankAccount` e `SavingsAccount`. Agora implementaremos a subclasse `CheckingAccount` para que você possa ver em detalhes como os métodos e campos de instância são herdados. Lembre-se de que a classe `BankAccount` tem três métodos e um campo de instância:

```
public class BankAccount
{
   public double getBalance() { . . . }
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }

   private double balance;
}
```

A classe `CheckingAccount` tem um método `deductFees` extra e um campo de instância `transactionCount` adicional, e sobrescreve os métodos `deposit` e `withdraw` para incrementar a contagem de transações:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   public void deductFees() { . . . }

   private int transactionCount;
}
```

Cada objeto da classe `CheckingAccount` tem dois campos de instância:

- `balance` (herdado de `BankAccount`)
- `transactionCount` (novo para `CheckingAccount`)

Você pode aplicar quatro métodos a objetos `CheckingAccount`:

- `getBalance()` (herdado de `BankAccount`)
- `deposit(double amount)` (sobrescreve o método `BankAccount`)

- `withdraw(double amount)` (sobrescreve o método `BankAccount`)
- `deductFees()` (novo para `CheckingAccount`)

Em seguida, vamos implementar esses métodos. O método `deposit` incrementa a contagem de transações e deposita o dinheiro:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Agora adicione amount a balance
      . . .
   }
   . . .
}
```

Temos um problema agora. Não podemos simplesmente adicionar `amount` a `balance`:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Agora adicione amount a balance
      balance = balance + amount; // Erro
   }
   . . .
}
```

Uma subclasse não tem acesso aos campos privados de sua superclasse.

Embora cada objeto `CheckingAccount` tenha um campo de instância `balance`, esse campo de instância é *privado* para a superclasse `BankAccount`. Os métodos da subclasse têm os mesmos direitos de acesso aos dados privados da superclasse que qualquer outro método tem. Se quiser modificar um campo privado da superclasse, você deverá utilizar um método público da superclasse.

Como é possível adicionar a quantia de depósito ao saldo, utilizando a interface pública da classe `BankAccount`? Há um método perfeito para esse propósito – a saber, o método `deposit` da classe `BankAccount`. Precisamos então invocar o método `deposit` a partir de algum objeto. Em qual objeto? A conta corrente em que o dinheiro é depositado – isto é, o parâmetro implícito do método `deposit` da classe `CheckingAccount`. Para invocar outro método no parâmetro implícito, você não especifica o parâmetro, basta escrever o nome do método assim:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Agora adicione amount a balance
      deposit(amount); // Incompleto
   }
   . . .
}
```

Mas isso não funcionará muito bem. O compilador interpreta

```
deposit(amount);
```

como

```
this.deposit(amount);
```

O parâmetro `this` é do tipo `CheckingAccount`. Há um método chamado `deposit` na classe `CheckingAccount`. Portanto, esse método será chamado – mas esse é exatamente o método que estamos escrevendo atualmente! O método chamará a ele próprio repetidamente e o programa trancará em uma recursão infinita (discutida no Capítulo 13).

Utilize a palavra-chave `super` para chamar um método da superclasse.

Em vez disso, devemos especificamente demonstrar que queremos invocar o método `deposit` da *superclasse*. Existe a palavra-chave especial `super` para esse propósito:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Agora adicione amount a balance
      super.deposit(amount);
   }
   . . .
}
```

Essa versão do método `deposit` está correta. Para depositar dinheiro em uma conta bancária, atualize a contagem de transações e chame o método `deposit` da superclasse.

Os métodos restantes são simples.

```
public class CheckingAccount extends BankAccount
{
   . . .
   public void withdraw(double amount)
   {
      transactionCount++;
      // Agora subtraia amount de balance
      super.withdraw(amount);
   }
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE
               * (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
   . . .
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
}
```

SINTAXE 10.2 Chamando um método de superclasse

`super.`*nomeDoMétodo*`(`*parâmetros*`);`

Exemplo:

```
public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Objetivo:

Chamar um método da superclasse em vez do método da classe atual

AUTOVERIFICAÇÃO DA APRENDIZAGEM

6. Por que o método `withdraw` da classe `CheckingAccount` chama `super.withdraw`?
7. Por que o método `deductFees` configura a contagem de transações como zero?

ERRO COMUM 10.2

Sombreando campos de instância

Uma subclasse não tem acesso aos campos de instância privados da superclasse. Por exemplo, os métodos da classe `CheckingAccount` não podem acessar o campo `balance`:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      balance = balance + amount; // Erro
   }
   . . .
}
```

É um erro comum de iniciante "resolver" esse problema adicionando um *outro* campo de instância com o mesmo nome.

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
```

```
        transactionCount++;
        balance = balance + amount;
    }
    . . .
    private double balance; // Não faça isso
}
```

Claro, agora o método `deposit` compila, mas não atualiza o saldo correto! Esse objeto `CheckingAccount` tem dois campos de instância, ambos chamados `balance` (ver a Figura 6). O método `getBalance` da superclasse recupera um deles e o método `deposit` da subclasse atualiza o outro

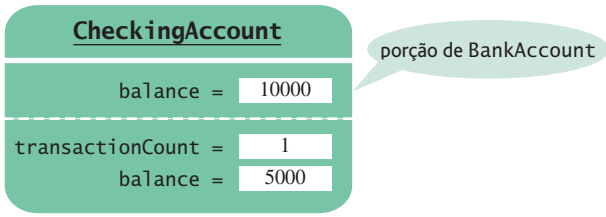


Figura 6 Sombreando campos de instância.

ERRO COMUM 10.3

Falhando ao invocar o método da superclasse

Um erro comum ao estender a funcionalidade de um método da superclasse é esquecer o qualificador `super`.. Por exemplo, para sacar dinheiro de uma conta corrente, atualize a contagem de transações e então retire a quantia:

```
public void withdraw(double amount)
{
   transactionCount++;
   withdraw(amount);
   // Erro – deveria ser super.withdraw(amount)
}
```

Aqui `withdraw(amount)` referencia o método `withdraw` aplicado ao parâmetro implícito do método. O parâmetro implícito é do tipo `CheckingAccount` e a classe `CheckingAccount` tem um método `withdraw`, de modo que este método será chamado. Naturalmente, isso chama o método atual mais uma vez, que chamará a ele mesmo de novo, repetidamente, até o programa ficar sem memória disponível. Em vez disso, você deve identificar precisamente qual método `withdraw` você quer chamar.

Um outro erro comum é simplesmente esquecer de chamar o método da superclasse. A funcionalidade da superclasse desaparece misteriosamente.

10.4 Construção das subclasses

Nesta seção, discutiremos a implementação de construtores nas subclasses. Como exemplo, vamos definir um construtor para configurar o saldo inicial de uma conta corrente.

Queremos invocar o construtor `BankAccount` para configurar o saldo como o saldo inicial. Há uma instrução especial para chamar o construtor da superclasse a partir de um construtor de subclasse. Utilize a palavra-chave `super`, seguida pelos parâmetros de construção entre parênteses:

```
public class CheckingAccount extends BankAccount
{
   public CheckingAccount(double initialBalance)
   {
      // Constrói a superclasse
      super(initialBalance);
      // Inicializa a contagem de transações
      transactionCount = 0;
   }
   . . .
}
```

Para chamar o construtor da superclasse, utilize a palavra-chave `super` como primeira instrução do construtor da subclasse.

Quando a palavra-chave `super` é seguida por um parêntese, ela indica uma chamada ao construtor da superclasse. Quando utilizada dessa maneira, a chamada de construtor deve ser *a primeira instrução do construtor da subclasse*. Por outro lado, se `super` for seguido por um ponto e por um nome de método, isso indica uma chamada a um método da superclasse, como visto na seção anterior. Essa chamada pode ser criada em qualquer lugar em todos os métodos da subclasse.

SINTAXE 10.3 Chamando um construtor da superclasse

```
especificadorDeAcesso NomeDaClasse(tipoDeParâmetro nomeDoParâmetro. . .)
{
   super(parâmetros);
   . . .
}
```

Exemplo:

```
public CheckingAccount(double initialBalance)
{
   super(initialBalance);
   transactionCount = 0;
}
```

Objetivo:

Invocar o construtor da superclasse. Observe que essa instrução deve ser a primeira instrução do construtor da subclasse.

O uso duplo da palavra-chave `super` é semelhante ao uso duplo da palavra-chave `this` (veja Tópico Avançado 3.1).

Se um construtor da subclasse não chamar o construtor da superclasse, a superclasse será construída com seu construtor padrão (isto é, o construtor que não tem nenhum parâmetro). Entretanto, se todos os construtores da superclasse exigirem parâmetros, o compilador então informará um erro.

Por exemplo, você pode implementar o construtor `CheckingAccount` sem chamar o construtor da superclasse. Desse modo, a classe `BankAccount` será construída com seu construtor padrão, que configura o saldo como zero. Naturalmente, o construtor `CheckingAccount` deve então depositar o saldo inicial de forma explícita.

O mais comum, entretanto, é os construtores da subclasse terem alguns parâmetros que eles passam para a superclasse e outros que eles utilizam para inicializar campos de subclasse.

Autoverificação da aprendizagem

8. Por que o construtor `SavingsAccount`, na Seção 10.1, não chamou seu construtor da superclasse?
9. Quando você invoca um método da superclasse com a palavra-chave `super`, a chamada tem de ser a primeira instrução do método da subclasse?

10.5 Convertendo subclasses em superclasses

Freqüentemente é necessário converter uma subclasse em uma superclasse. Ocasionalmente, você precisa executar a conversão na direção contrária. Esta seção discute as regras de conversão.

Referências a subclasse podem ser convertidas em referências da superclasse.

A classe `SavingsAccount` estende a classe `BankAccount`. Em outras palavras, um objeto `SavingsAccount` é um caso especial de um objeto `BankAccount`. Portanto, uma referência a um objeto `SavingsAccount` pode ser convertida em uma referência a `BankAccount`.

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
```

Além disso, todas as referências podem ser convertidas no tipo `Object`.

```
Object anObject = collegeFund;
```

Agora as três referências a objeto armazenadas em `collegeFund`, `anAccount` e `anObject` referenciam o mesmo objeto do tipo `SavingsAccount` (veja Figura 7).

Mas a referência a objeto `anAccount` só conhece parcialmente a história do objeto que ela referencia. Como `anAccount` é um objeto do tipo `BankAccount`, você pode utilizar os métodos `deposit` e `withdraw` para alterar o saldo da conta poupança. Você não pode, porém, utilizar o método `addInterest` – ele não é um método da superclasse `BankAccount`:

```
anAccount.deposit(1000); // OK
anAccount.addInterest();
   // Não – não é um método da classe à qual anAccount pertence
```

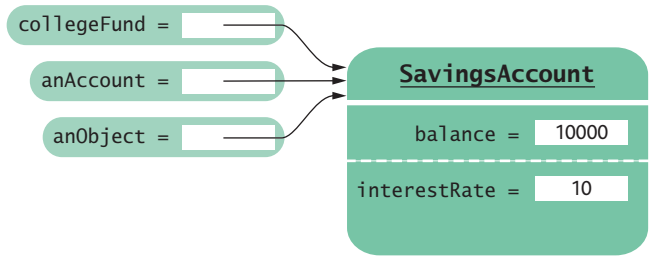


Figura 7 Variáveis de diferentes tipos referenciam o mesmo objeto.

E, naturalmente, a variável `anObject` conhece menos ainda. Você nem mesmo pode aplicar o método `deposit` a ela – `deposit` não é um método da classe `Object`.

A conversão de referências é diferente de uma conversão numérica, como uma conversão de um inteiro em um número de ponto flutuante. Se converter um inteiro, digamos 4, no valor `double` 4.0, então a representação modificará: o valor `double` 4.0 usa uma seqüência diferente de bits que o valor `int` 4. Mas, quando você converte uma referência `SavingsAccount` em uma referência `BankAccount`, o valor da referência permanece o mesmo – ele é a posição do objeto na memória. Mas, depois da conversão, há menos informações sobre o objeto. Sabemos apenas que ele é uma conta bancária. Talvez ele seja uma conta bancária simples, uma conta poupança ou outro tipo de conta bancária.

Por que alguém iria *querer* conhecer menos sobre um objeto e armazenar uma referência em um campo de objeto de uma superclasse? Isso pode acontecer se você quiser *reutilizar o código* que conhece a superclasse, mas não a subclasse. Eis um exemplo típico. Considere um método `transfer` que transfere dinheiro de uma conta para outra:

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Você pode utilizar esse método para transferir dinheiro de uma conta bancária para outra:

```
BankAccount momsAccount = . . . ;
BankAccount harrysAccount = . . . ;
momsAccount.transfer(1000, harrysAccount);
```

Você *também* pode utilizar esse método para transferir dinheiro para uma `CheckingAccount`:

```
CheckingAccount harrysChecking = . . . ;
momsAccount.transfer(1000, harrysChecking);
   // OK passar uma referência a CheckingAccount para um método que espera uma
   // BankAccount
```

O método `transfer` espera uma referência a uma `BankAccount` e obtém uma referência à subclasse `CheckingAccount`. Felizmente, em vez de reclamar de uma não-correspondência de tipo, o compilador simplesmente copia a referência de subclasse `harrysChecking` à referência da superclasse `other`. Na verdade, o método `transfer` não sabe disso, nesse

caso, `other` referencia uma `CheckingAccount`. Ele só sabe que `other` é uma `BankAccount` e não precisa saber nada mais. Só interessa que o objeto `other` possa executar o método `deposit`.

Bem ocasionalmente, você precisará executar a conversão oposta, de uma referência da superclasse para uma referência da subclasse. Por exemplo, você pode ter uma variável do tipo `Object` e saber que ela realmente contém uma referência a `BankAccount`. Nesse caso, você pode utilizar uma coerção, ou typecasting, para converter o tipo:

```
BankAccount anAccount = (BankAccount) anObject;
```

Mas essa coerção é um pouco perigosa. Se você estiver errado e `anObject`, na verdade, referir-se a um objeto de um tipo não-relacionado, uma exceção será lançada.

O operador `instanceof` testa se um objeto pertence a um tipo particular.

Para se proteger de coerções ruins, você pode utilizar o operador `instanceof`. Ele testa se um objeto pertence a um tipo específico. Por exemplo,

```
anObject instanceof BankAccount
```

retorna `true` se o tipo de `anObject` for conversível em `BankAccount`. Isso acontece se `anObject` referir-se a um `BankAccount` real ou a uma subclasse como `SavingsAccount`. Utilizando o operador `instanceof`, uma coerção segura pode ser programada desta maneira:

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

SINTAXE 10.4 O operador `instanceof`

objeto `instanceof` *NomeDoTipo*

Exemplo:

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Objetivo:

Retornar `true` se o *objeto* for uma instância de *NomeDoTipo* (ou um dos seus subtipos) e `false` caso contrário

AUTOVERIFICAÇÃO DA APRENDIZAGEM

10. Por que o segundo parâmetro do método `transfer` tem de ser do tipo `BankAccount` e não, por exemplo, `SavingsAccount`?
11. Por que não podemos modificar o segundo parâmetro do método `transfer` para o tipo `Object`?

10.6 Polimorfismo

Em Java, o tipo de uma variável não determina completamente o tipo do objeto a que ela se refere. Por exemplo, uma variável do tipo `BankAccount` pode armazenar uma referência a um objeto `BankAccount` real ou a um objeto de alguma subclasse como `SavingsAccount`. Já vimos esse fenômeno no Capítulo 9, com variáveis cujos tipos eram uma interface. Uma variável cujo tipo é `Measurable` contém uma referência a um objeto de uma classe que implementa a interface `Measurable`, talvez um objeto `Coin` ou um objeto de uma classe completamente diferente.

O que acontece quando você invoca um método? Por exemplo,

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
```

Qual método de depósito é chamado? O parâmetro `anAccount` é do tipo `BankAccount`, portanto, parece que `BankAccount.deposit` é chamado. Por outro lado, a classe `CheckingAccount` fornece seu próprio método `deposit` que atualiza a contagem de transações. O campo `anAccount`, na verdade, referencia um objeto da subclasse `Checking-Account`, assim seria apropriado se, em vez disso, o método `CheckingAccount.deposit` fosse chamado.

Em Java, chamadas de método *sempre são determinadas pelo tipo do objeto real*, não pelo tipo da referência ao objeto. Isto é, se o objeto real for do tipo `CheckingAccount`, então o método `CheckingAccount.deposit` será chamado. Não importa se a referência ao objeto seja armazenada em um campo do tipo `BankAccount`. Como discutido no Capítulo 9, a capacidade de referenciar objetos de múltiplos tipos com comportamento variado chama-se polimorfismo.

Se o polimorfismo é tão poderoso, por que não armazenar todas as referências a conta em variáveis do tipo `Object`? Isso não funciona porque o compilador precisa verificar que somente os métodos válidos são invocados. O tipo `Object` não define um método `deposit` – o tipo `BankAccount` (pelo menos) é necessário para fazer uma chamada ao método `deposit`.

Reexamine o método `transfer` para ver o polimorfismo em ação. Eis a implementação do método:

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Suponha que você chame

```
anAccount.transfer(1000, anotherAccount);
```

Duas chamadas de método são o resultado

```
anAccount.withdraw(1000);
anotherAccount.deposit(1000);
```

Dependendo dos tipos reais de `anAccount` e `anotherAccount`, versões diferentes dos métodos `withdraw` e `deposit` são chamadas.

Se examinar a implementação do método `transfer`, talvez não seja imediatamente óbvio que a primeira chamada de método

```
withdraw(amount);
```

depende do tipo de um objeto. Mas essa chamada é um atalho para

```
this.withdraw(amount);
```

O parâmetro `this` contém uma referência ao parâmetro implícito, que pode referenciar um objeto `BankAccount` ou um objeto de uma subclasse.

O programa a seguir chama os métodos polimórficos `withdraw` e `deposit`. Você deve calcular manualmente o que o programa deve imprimir para cada saldo bancário e confirmar se os métodos corretos foram de fato chamados.

ch10/accounts/AccountTester.java

```
/**
   Este programa testa a classe BankAccount e
   suas subclasses.
*/
public class AccountTester
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings
            = new SavingsAccount(0.5);

      CheckingAccount harrysChecking
            = new CheckingAccount(100);

      momsSavings.deposit(10000);

      momsSavings.transfer(2000, harrysChecking);
      harrysChecking.withdraw(1500);
      harrysChecking.withdraw(80);

      momsSavings.transfer(1000, harrysChecking);
      harrysChecking.withdraw(400);

      // Simula o fim do mês
      momsSavings.addInterest();
      harrysChecking.deductFees();

      System.out.println("Mom's savings balance: "
            + momsSavings.getBalance());
      System.out.println("Expected: 7035");

      System.out.println("Harry's checking balance: "
            + harrysChecking.getBalance());
      System.out.println("Expected: 1116");
   }
}
```

ch10/accounts/BankAccount.java

```
/**
   Uma conta bancária tem um saldo que pode ser alterado por
   depósitos e saques.
*/
public class BankAccount
{
   /**
      Cria uma conta bancária com saldo zero.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Constrói uma conta bancária com um saldo especificado.
      @param initialBalance saldo inicial
   */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }

   /**
      Deposita dinheiro na conta bancária.
      @param amount valor a depositar
   */
   public void deposit(double amount)
   {
      balance = balance + amount;
   }

   /**
      Retira dinheiro da conta bancária.
      @param amount valor a retirar
   */
   public void withdraw(double amount)
   {
      balance = balance - amount;
   }

   /**
      Obtém o saldo atual da conta bancária.
      @return saldo atual
   */
   public double getBalance()
   {
      return balance;
   }

   /**
      Transfere dinheiro da conta bancária para outra conta.
      @param amount quantia a transferir
```

```
54         @param other outra conta
55      */
56      public void transfer(double amount, BankAccount other)
57      {
58         withdraw(amount);
59         other.deposit(amount);
60      }
61
62      private double balance;
63   }
```

ch10/accounts/CheckingAccount.java

```
 1   /**
 2      Uma conta corrente que cobra tarifas por transação.
 3   */
 4   public class CheckingAccount extends BankAccount
 5   {
 6      /**
 7         Constrói uma conta corrente com um dado saldo.
 8         @param initialBalance o saldo inicial
 9      */
10      public CheckingAccount(double initialBalance)
11      {
12         // Constrói a superclasse
13         super(initialBalance);
14
15         // Inicializa a contagem de transações
16         transactionCount = 0;
17      }
18
19      public void deposit(double amount)
20      {
21         transactionCount++;
22         // Agora adicione amount balance
23         super.deposit(amount);
24      }
25
26      public void withdraw(double amount)
27      {
28         transactionCount++;
29         // Agora subtraia amount de balance
30         super.withdraw(amount);
31      }
32
33      /**
34         Subtrai as tarifas acumuladas e redefine a
35         contagem de transações.
36      */
37      public void deductFees()
38      {
39         if (transactionCount > FREE_TRANSACTIONS)
40         {
41            double fees = TRANSACTION_FEE *
42                  (transactionCount - FREE_TRANSACTIONS);
```

```
43            super.withdraw(fees);
44         }
45         transactionCount = 0;
46      }
47
48      private int transactionCount;
49
50      private static final int FREE_TRANSACTIONS = 3;
51      private static final double TRANSACTION_FEE = 2.0;
52   }
```

ch10/accounts/SavingsAccount.java

```
1   /**
2      Uma conta que rende juros a uma taxa fixa.
3   */
4   public class SavingsAccount extends BankAccount
5   {
6      /**
7         Constrói uma conta poupança com uma dada taxa de juros.
8         @param rate taxa de juros
9      */
10     public SavingsAccount(double rate)
11     {
12        interestRate = rate;
13     }
14
15     /**
16        Adiciona os juros ganhos de acordo com o saldo da conta.
17     */
18     public void addInterest()
19     {
20        double interest = getBalance() * interestRate / 100;
21        deposit(interest);
22     }
23
24     private double interestRate;
25  }
```

Saída

```
Mom's savings balance: 7035.0
Expected: 7035
Harry's checking balance: 1116.0
Expected: 1116
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

12. Se a for uma variável do tipo `BankAccount` que contém uma referência não-`null`, o que você sabe a respeito do objeto que a referencia?

13. Se a referencia uma conta corrente, qual é o efeito de chamar `a.transfer(1000, a)`?

Tópico avançado 10.1

Classes abstratas

O Tópico Avançado 10.1 introduz o conceito de classes e métodos abstratos. Um método abstrato não tem implementação. (Todos os métodos de uma interface são automaticamente abstratos.) Você não pode construir objetos das classes abstratas, em geral porque a classe tem um ou mais métodos abstratos. Mas há uma diferença importante entre classes abstratas e interfaces – elas podem ter campos de instância e também métodos e construtores concretos.

Tópico avançado 10.2

Métodos e classes final

O Tópico Avançado 10.2 discute classes e métodos final. Um método final não pode ser redefinido em uma subclasse. Uma classe final não pode dividida em subclasses.

10.7 Controle de acesso

Java tem quatro níveis de controle de acesso a campos, métodos e classes:

- acesso `public`
- acesso `private`
- acesso `protected` (veja Tópico Avançado 10.3)
- acesso de pacote (o padrão, quando nenhum modificador de acesso é fornecido)

Já utilizamos extensamente os modificadores `private` e `public`. Recursos privados só podem ser acessados pelos métodos da sua própria classe. Recursos públicos podem ser acessados por métodos de todas as classes. Discutiremos o acesso protegido no Tópico Avançado 10.3 – não precisaremos dele neste livro.

Um campo ou método que não é declarado como `public`, `private` ou `protected` pode ser acessado por todas as classes do mesmo pacote, o que normalmente não é desejável.

Se você não fornecer um modificador de controle de acesso, então o padrão será *acesso de pacote*. Isto é, todos os métodos das classes no mesmo pacote acessam o recurso. Por exemplo, se uma classe for declarada como `public`, todas as outras classes em todos os pacotes poderão utilizá-la. Mas se uma classe for declarada sem um modificador de acesso, somente as outras classes do mesmo pacote poderão utilizá-la. O acesso de pacote é um bom padrão para classes, mas é extremamente ruim para campos. Campos de instância e campos estáticos de classes sempre devem ser `private`. Há algumas exceções:

- Constantes públicas (campos `public static final`) são úteis e seguras.
- Alguns objetos, como `System.out`, precisam estar acessíveis a todos os programas e, portanto, devem ser públicos.
- Ocasionalmente, várias classes em um pacote devem colaborar bem estreitamente. Nesse caso, talvez faça sentido fornecer algum acesso de pacote aos campos. Mas classes internas normalmente são uma solução melhor – vimos exemplos no Capítulo 9.

É um erro comum *esquecer* a palavra-chave `private`, abrindo assim uma potencial brecha de segurança. Por exemplo, no momento em que escrevíamos este livro, a classe `Window` no pacote `java.awt` continha a seguinte declaração:

```
public class Window extends Container
{
   String warningString;
   . . .
}
```

O programador foi descuidado e não tornou o campo privado. Na verdade, não havia uma boa razão para conceder acesso de pacote ao campo `warningString` – nenhuma outra classe o acessa. É um risco de segurança. Pacotes não são entidades fechadas – qualquer programador pode criar uma nova classe, adicioná-la ao pacote `java.awt` e ganhar acesso aos campos `warningString` de todos os objetos `Window`! (Na realidade, essa possibilidade incomodava tanto os implementadores Java que versões recentes da máquina virtual recusam-se a carregar classes desconhecidas cujo nome de pacote inicia com "`java.`". Mas os pacotes que você cria não desfrutam dessa proteção.)

O acesso de pacote a campos raramente é útil, e a maioria dos campos recebe acidentalmente acesso de pacote porque o programador simplesmente esqueceu da palavra-chave `private`.

Em geral, métodos devem ser `public` ou `private`. Recomendamos evitar o uso de métodos visíveis a pacotes.

Classes e interfaces podem ter acesso público ou de pacote. Classes que geralmente são úteis devem ter acesso público. Classes utilizadas por razões de implementação devem ter acesso de pacote. Você pode ocultá-las ainda melhor transformando-as em classes internas; vimos exemplos de classes internas no Capítulo 9. Há alguns exemplos de classes internas públicas, como a classe `Ellipse2D.Double`, que discutimos no Capítulo 2 (Seção 2.13). Mas, em geral, classes internas não devem ser públicas.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

14. Qual é uma razão comum para definir campos de instância visíveis no pacote?

15. Se uma classe com um construtor público tiver acesso de pacote, quem pode construir objetos dela?

ERRO COMUM 10.4

Acesso acidental a pacote

É muito fácil esquecer do modificador `private` em campos de instância.

```
public class BankAccount
{
   . . .
   double balance; // Acesso de pacote realmente intencional?
}
```

Isso deve ter sido apenas um descuido. O programador provavelmente nunca pretendeu conceder acesso para esse campo a outras classes no mesmo pacote. O compilador não reclamará, naturalmente. Bem mais tarde, algum outro programador poderá tirar proveito do privilégio de acesso, por conveniência ou com má intenção. Esse é um problema sério e você deve ter o hábito de verificar a ausência de modificadores `private` nas suas declarações de campo.

ERRO COMUM 10.5

Tornando métodos herdados menos acessíveis

Se uma superclasse declarar um método como sendo publicamente acessível, você não poderá mais sobrescrevê-lo para ser privado. Por exemplo,

```
public class BankAccount
{
   public void withdraw(double amount) { . . . }
   . . .
}
public class CheckingAccount extends BankAccount
{
   private void withdraw(double amount) { . . . }
      // Erro – o método da subclasse não pode mais ser privado
   . . .
}
```

O compilador não permite isso, porque a privacidade aumentada seria uma ilusão. Qualquer pessoa ainda pode chamar o método por meio de uma referência da superclasse:

```
BankAccount account = new CheckingAccount();
account.withdraw(100000); // Chama CheckingAccount.withdraw
```

Por causa do polimorfismo, o método da subclasse é chamado.

Esses erros normalmente são um descuido. Se esquecer do modificador `public`, seu método de subclasse terá acesso de pacote, o que é mais restritivo. Simplesmente restaure o modificador `public` e o erro desaparecerá.

TÓPICO AVANÇADO 10.3

Acesso protegido

O Tópico Avançado 10.3 discute o especificador de acesso `protected`. Um campo ou método protegido pode ser acessado por todas as subclasses e por todas as classes no mesmo pacote.

10.8 `Object`: a superclasse cósmica

Em Java, cada classe definida sem uma cláusula `extends` explícita estende automaticamente a classe `Object`. Isto é, a classe `Object` é a superclasse direta ou indireta de *toda* classe em Java (veja Figura 8).

Naturalmente, os métodos da classe `Object` são muito gerais. Eis os mais úteis:

Método	Propósito
`String toString()`	Retorna uma representação de string do objeto
`boolean equals(Object otherObject)`	Testa se o objeto é igual a outro objeto
`Object clone()`	Cria uma cópia completa de um objeto

Uma boa idéia é sobrescrever esses métodos nas suas classes.

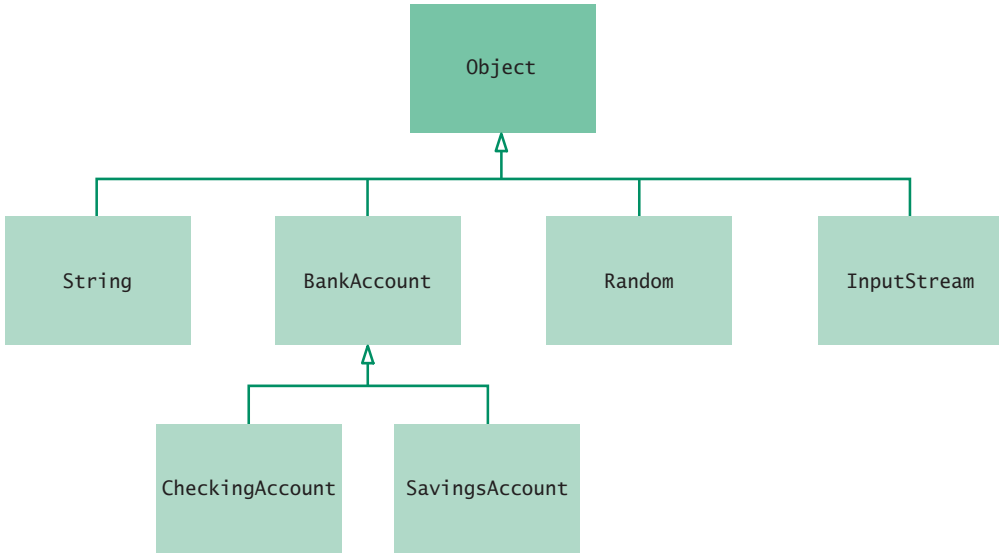


Figura 8 A classe `Object` é a superclasse de todas as classes em Java.

10.8.1 Sobrescrevendo o método `toString`

Defina o método `toString` para produzir uma string que descreve o estado do objeto.

O método `toString` retorna uma representação de string para cada objeto. Ele é utilizado para depuração. Por exemplo,

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
  // Configura s como "java.awt.Rectangle[x=5,y=10,width=
     20,height=30]"
```

Na verdade, esse método `toString` é chamado sempre que você concatena uma string com um objeto. Considere a concatenação

```
"box=" + box;
```

Em um dos lados do operador de concatenação `+` há uma string, mas no outro lado há uma referência a objeto. O compilador Java invoca automaticamente o método `toString` para transformar o objeto em uma string. As duas strings são então concatenadas. Nesse caso, o resultado é a string:

```
"box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

O compilador pode invocar o método `toString`, porque ele sabe que *todo* objeto tem um método `toString`: todas as classes estendem a classe `Object` e essa classe define `toString`.

Como você sabe, números também são convertidos em strings quando são concatenados com outras strings. Por exemplo,

```
int age = 18;
String s = "Harry's age is " + age;
   // Configura s como "Harry's age is 18"
```

Nesse caso, o método `toString` não está envolvido. Números não são objetos e não há nenhum método `toString` para eles. Há, porém, apenas um pequeno conjunto de tipos primitivos e o compilador sabe como convertê-los em strings.

Vamos testar o método `toString` para a classe `BankAccount`:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
   // Configura para algo como"BankAccount@d24606bf"
```

Isso é decepcionante – tudo o que é impresso é o nome da classe, seguido pelo código de hash, um código aparentemente aleatório. O código de hash pode ser utilizado para informar sobre outros objetos – objetos diferentes que provavelmente têm códigos de hash também diferentes. (Veja o Capítulo 16 para detalhes.)

O código de hash não nos interessa. Queremos saber o que está dentro do objeto. Mas, naturalmente, o método `toString` da classe `Object` não sabe o que há dentro da classe `BankAccount`. Portanto, temos de sobrescrever o método e fornecer uma versão própria na classe `BankAccount`. Seguiremos o mesmo formato do método `toString` da classe `Rectangle` que: primeiro imprime o nome da classe e então os valores dos campos de instância dentro de colchetes.

```
public class BankAccount
{
   . . .
   public String toString()
   {
      return "BankAccount[balance=" + balance + "]";
   }
}
```

Isso funciona melhor:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
   // Configura s como "BankAccount[balance=5000]"
```

DICA DE PRODUTIVIDADE 10.1

Fornecer `toString` em todas as classes

Se você tiver uma classe cujo método `toString()` retorna uma string que descreve o estado do objeto, você então poderá simplesmente chamar `System.out.println(x)` sempre que precisar inspecionar o estado atual de um objeto `x`. Isso funciona porque o método `println` da classe `PrintStream` invoca `x.toString()` quando ele precisa imprimir um objeto, o que é extremamente útil quando há um erro no seu programa e os objetos não se comportam da maneira esperada. Você pode simplesmente inserir algumas instruções de impressão e examinar rapidamente dentro do estado do objeto durante a execução do programa. Alguns depuradores podem até mesmo invocar o método `toString` nos objetos que você inspeciona.

Certamente é um pouco mais difícil escrever um método `toString` quando você não está seguro se o programa precisa de um – afinal de contas, talvez ele funcione corretamente na primeira tentativa. Mais uma vez, muitos programas não funcionam na primeira tentativa. Assim que você descobrir que ele não funciona corretamente, considere adicionar esses métodos `toString` para ajudá-lo a depurar o programa.

TÓPICO AVANÇADO 10.4

Herança e o método `toString`

O Tópico Avançado 10.4 fornece uma receita de como implementar o método `toString` para que ele possa ser facilmente estendido nas subclasses.

10.8.2 Sobrescrevendo o método `equals`

Defina o método `equals` para testar se dois objetos têm um estado igual.

O método `equals` é chamado sempre que você desejar comparar se dois objetos contêm o mesmo conteúdo:

```
if (coin1.equals(coin2)) . . .
   // O conteúdo é o mesmo – veja Figura 9
```

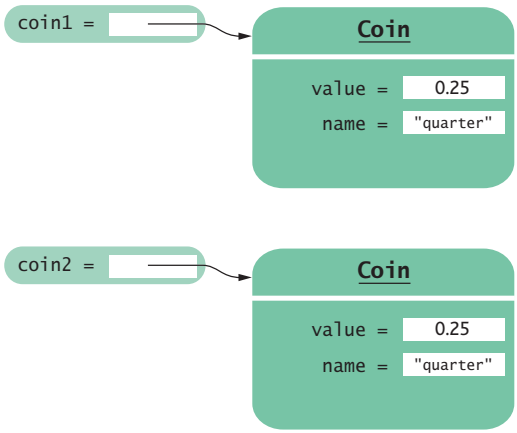


Figura 9 Duas referências a objetos idênticos.

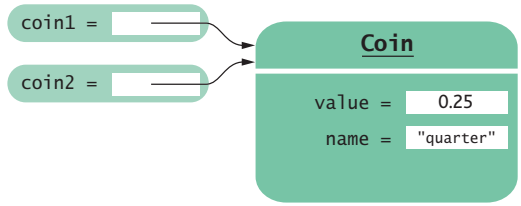


Figura 10 Duas referências ao mesmo objeto.

Isso é diferente do teste com o operador ==, que testa se as duas referências apontam para o *mesmo objeto*:

```
if (coin1 == coin2) . . .
   // Os objetos são os mesmos – veja Figura 10
```

Vamos implementar o método `equals` para a classe `Coin`. Você precisa sobrescrever o método `equals` da classe `Object`:

```
public class Coin
{
   . . .
   public boolean equals(Object otherObject)
   {
      . . .
   }
   . . .
}
```

Agora temos um pequeno problema. A classe `Object` não sabe nada sobre moedas, portanto ela define o parâmetro `otherObject` do método `equals` como tendo o tipo `Object`. Ao redefinir o método, você perde a permissão de alterar a assinatura do objeto. Faça uma coerção no parâmetro para a classe `Coin`:

```
Coin other = (Coin) otherObject;
```

Você pode então comparar as duas moedas.

```
public boolean equals(Object otherObject)
{
   Coin other = (Coin) otherObject;
   return name.equals(other.name)
         && value == other.value;
}
```

Observe que você precisa utilizar `equals` para comparar campos de objetos, mas utilize `==` para comparar campos numéricos.

Ao sobrescrever o método `equals`, você também deve sobrescrever o método `hashCode` para que objetos idênticos tenham o mesmo código de hash – consulte o Capítulo 16 para obter detalhes.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

16. A chamada `x.equals(x)` sempre deve retornar `true`?
17. Você pode implementar `equals` em termos do `toString`? Você deveria?

ERRO COMUM 10.6

Definindo o método `equals` com o tipo errado de parâmetro

Considere a seguinte versão, aparentemente mais simples, do método `equals` para a classe `Coin`:

```
public boolean equals(Coin other) // Não faça isso!
{
   return name.equals(other.name) && value == other.value;
}
```

Aqui, o parâmetro do método `equals` tem o tipo `Coin`, não `Object`.

Infelizmente, esse método *não sobrescreve* o método `equals` na classe `Object`. Em vez disso, a classe `Coin` agora tem dois métodos `equals` diferentes:

```
boolean equals(Coin other) // Definido na classe Coin
boolean equals(Object otherObject) // Herdado da classe Object
```

Isso é propenso a erro porque o método `equals` errado pode ser chamado. Por exemplo, considere estas definições de variáveis:

```
Coin aCoin = new Coin(0.25, "quarter");
Object anObject = new Coin(0.25, "quarter");
```

A chamada `aCoin.equals(anObject)` chama o segundo método `equals`, que retorna `false`.

O correto é assegurar que você utilize os tipos `Object` para o parâmetro explícito do método `equals`.

TÓPICO AVANÇADO 10.5

Herança e o método `equals`

O Tópico Avançado 10.5 analisa os problemas sutis que surgem quando o método `equals` é redefinido em uma subclasse e fornece uma receita de como minimizar esses problemas.

10.8.3 O método clone

Sabemos que copiar uma referência a objeto simplesmente resulta em duas referências ao mesmo objeto:

```
BankAccount account = new BankAccount(1000);
BankAccount account2 = account;
account2.deposit(500);
   // Agora account e account2 referenciam uma conta bancária com um saldo de 1500
```

O método clone cria um novo objeto com o mesmo estado de um objeto existente.

O que se pode você fazer se realmente deseja-se criar uma cópia de um objeto? Esse é o propósito do método `clone`. O método `clone` deve retornar um *novo* objeto que tem um estado idêntico ao objeto existente (veja Figura 11).

Implementar o método `clone` é bem mais difícil do que implementar os métodos `toString` ou `equals` – veja Tópico Avançado 10.6 para detalhes.

Vamos supor que alguém implementou o método `clone` para a classe `BankAccount`. Eis como chamá-lo:

```
BankAccount clonedAccount = (BankAccount) account.clone();
```

O tipo de retorno do método `clone` é a classe `Object`. Ao chamar o método, você deve utilizar uma coerção para convencer o compilador de que `account.clone()` é realmente do mesmo tipo de `clonedAccount`.

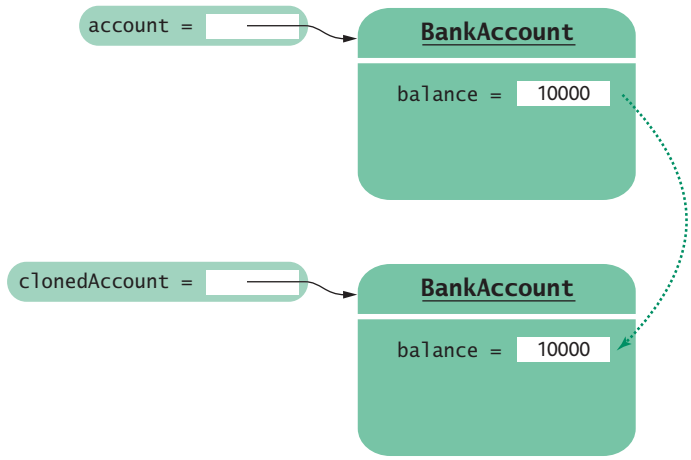


Figura 11
Clonando objetos.

ERRO COMUM 10.7

Esquecendo de clonar

Em Java, campos de objeto contêm referências a objetos, não objetos reais. Isso pode ser conveniente para atribuir *dois nomes ao mesmo objeto:*

```
BankAccount harrysChecking = new BankAccount();
BankAccount slushFund = harrysChecking;
    // Utilize a conta bancária de Harry para o caixa dois
slushFund.deposit(80000);
    // Uma grande soma de dinheiro acaba na conta bancária de Harry
```

Mas, se você não pretende que duas referências referenciem o mesmo objeto, então isso é um problema. Nesse caso, você deve utilizar o método `clone`:

```
BankAccount slushFund = (BankAccount) harrysChecking.clone();
```

DICA DE QUALIDADE 10.1

Clone campos de instância mutáveis em métodos de acesso

A Dica de Qualidade 10.1 sugere que seus métodos de acesso não devem revelar referências a campos de instância modificáveis, mas que os valores de campo de instância devem ser primeiro clonados.

TÓPICO AVANÇADO 10.6

Implementando o método `clone`

O Tópico Avançado 10.6 explica como implementar o método `clone` para suas próprias classes.

TÓPICO AVANÇADO 10.7

Tipos enumerados revistos

O Tópico Avançado 10.7 revê os tipos enumerados e explica que todos são subclasses das classes `Enum`. A classe `Enum` tem definições adequadas dos métodos `toString`, `equals` e `clone`, que são herdados por todos os tipos enumerados.

FATO ALEATÓRIO 10.1

Linguagens de criação de scripts

O Fato Aleatório 10.1 discute as linguagens de criação de scripts projetadas para desenvolvimento rápido, com uma estrutura simples e menos regras de sintaxe, freqüentemente dando suporte a um aplicativo específico (como software para escritório ou um navegador Web).

10.9 Utilizando herança para personalizar frames

Defina uma subclasse `JFrame` para um frame complexo.

À medida que você adiciona mais componentes de interface a um frame, este pode tornar-se bem complexo. Seus programas serão mais fáceis de entender quando você utilizar a herança para frames complexos.

Projete uma subclasse de `JFrame`. Armazene os componentes como campos de instância. Inicialize-os no construtor da sua subclasse. Se o código de inicialização tornar-se complexo, simplesmente adicione alguns métodos auxiliares.

Aqui, executamos esse processo para o programa visualizador de investimentos do Capítulo 9.

```
public class InvestmentFrame extends JFrame
{
   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);

      // Usa campos de instância para os componentes
      label = new JLabel("balance: " + account.getBalance());

      // Usa métodos auxiliares
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createButton()
   {
      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);
      button = new JButton("Add Interest");
   }

   private void createPanel()
   {
      panel = new JPanel();
      panel.add(button);
      panel.add(label);
      add(panel);
   }

   private JButton button;
   private JLabel label;
   private JPanel panel;
   private BankAccount account;
}
```

Essa abordagem difere dos programas do Capítulo 9. Naqueles programas, simplesmente configuramos o frame no método `main` de uma classe visualizadora.

É um pouco mais trabalhoso fornecer uma classe separada para o frame. Mas a classe do frame torna mais fácil organizar o código que constrói o elemento da interface com o usuário. Utilizaremos essa abordagem para todos os exemplos deste capítulo.

Naturalmente, ainda precisamos de uma classe com um método `main`:

```
public class InvestmentViewer2
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

18. Quantos arquivos-fonte Java são solicitados pela aplicação visualizadora de investimentos quando utilizamos a herança para definir a classe do frame?

19. Por que o construtor `InvestmentFrame` chama `setSize(FRAME_WIDTH, FRAME_HEIGHT)`, enquanto o método `main` da classe visualizadora de investimentos no Capítulo 9 chama `frame.setSize(FRAME_WIDTH, FRAME_HEIGHT)`?

TÓPICO AVANÇADO 10.8

Adicionando o método `main` à classe de frame

Reexamine as classes `InvestmentFrame` e `InvestmentViewer2`. Alguns programadores preferem combinar essas duas classes, adicionando o método `main` à classe de frame:

```
public class InvestmentFrame extends JFrame
{
   public static void main(String[] args)
   {
      JFrame frame = new InvestmentFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }

   public InvestmentFrame()
   {
      account = new BankAccount(INITIAL_BALANCE);

      // Usa campos de instância para os componentes
      label = new JLabel("balance: " + account.getBalance());

      // Usa métodos auxiliares
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }
   . . .
}
```

Isso é um atalho conveniente que você encontrará em muitos programas, mas ele confunde as responsabilidades entre a classe de frame e o programa. Portanto, não utilizamos essa abordagem neste livro.

10.10 Processando entrada de texto

Uma aplicação gráfica pode receber entrada de texto chamando o método `showInputDialog` da classe `JOptionPane`, mas abrir uma caixa de diálogo separada para cada entrada não é uma interface natural com o usuário. A maioria dos programas gráficos coleta a entrada de texto por meio de campos de texto (ver a Figura 12). Nesta seção, você aprenderá a adicionar campos de texto a uma aplicação gráfica e ler o que o usuário digita nelas.

Utilize componentes `JTextField` para fornecer espaço para a entrada do usuário. Insira um `JLabel` ao lado de cada campo de texto.

A classe `JTextField` fornece um campo de texto. Ao construir um campo de texto, você precisa fornecer a largura – o número aproximado de caracteres que você espera que o usuário digite.

```
final int FIELD_WIDTH = 10;
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

Os usuários podem digitar caracteres adicionais, mas nesse caso uma parte do conteúdo do campo torna-se invisível.

É recomendável rotular cada campo de texto para que o usuário saiba o que digitar nele. Construa um objeto `JLabel` para cada rótulo:

```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

Você quer dar ao usuário a oportunidade de digitar todas as informações nos campos de texto antes de processá-las. Portanto, você deve fornecer um botão que o usuário possa pressionar para indicar que a entrada está pronta para processamento.

Quando esse botão é clicado, seu método `actionPerformed` lê a entrada de usuário a partir dos campos de texto, utilizando o método `getText` da classe `JTextField`. O método `getText` retorna um objeto `String`. No nosso programa de exemplo, transformamos a string em um número, utilizando o método `Double.parseDouble`:

```
class AddInterestListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      double rate = Double.parseDouble(rateField.getText());
      . . .
   }
}
```

A seguinte aplicação é um protótipo útil para uma interface gráfica com o usuário para cálculos arbitrários. Você pode modificá-lo facilmente para suas próprias necessidades. Adicione outros componentes de entrada ao frame. Altere o conteúdo do método `actionPerformed` para executar outros cálculos. Exiba o resulta em um rótulo.

Figura 12 Aplicação com um campo de texto.

ch10/textfield/InvestmentViewer3.java

```
1  import javax.swing.JFrame;
2
3  /**
4     Esse programa exibe o rendimento de um investimento.
5  */
6  public class InvestmentViewer3
7  {
8     public static void main(String[] args)
9     {
10       JFrame frame = new InvestmentFrame();
11       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12       frame.setVisible(true);
13    }
14 }
```

ch10/textfield/InvestmentFrame.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6  import javax.swing.JPanel;
7  import javax.swing.JTextField;
8
9  /**
10    Um frame que mostra o crescimento de um investimento com juros variáveis.
11 */
12 public class InvestmentFrame extends JFrame
13 {
14    public InvestmentFrame()
15    {
16       account = new BankAccount(INITIAL_BALANCE);
17
18       // Usa campos de instância para os componentes
19       resultLabel = new JLabel("balance: " + account.getBalance());
20
21       // Usa métodos auxiliares
22       createTextField();
23       createButton();
24       createPanel();
25
26       setSize(FRAME_WIDTH, FRAME_HEIGHT);
27    }
28
29    private void createTextField()
30    {
31       rateLabel = new JLabel("Interest Rate: ");
32
33       final int FIELD_WIDTH = 10;
34       rateField = new JTextField(FIELD_WIDTH);
35       rateField.setText("" + DEFAULT_RATE);
36    }
```

```
37
38      private void createButton()
39      {
40         button = new JButton("Add Interest");
41
42         class AddInterestListener implements ActionListener
43         {
44            public void actionPerformed(ActionEvent event)
45            {
46               double rate = Double.parseDouble(
47                     rateField.getText());
48               double interest = account.getBalance()
49                     * rate / 100;
50               account.deposit(interest);
51               resultLabel.setText(
52                     "balance: " + account.getBalance());
53            }
54         }
55
56         ActionListener listener = new AddInterestListener();
57         button.addActionListener(listener);
58      }
59
60      private void createPanel()
61      {
62         panel = new JPanel();
63         panel.add(rateLabel);
64         panel.add(rateField);
65         panel.add(button);
66         panel.add(resultLabel);
67         add(panel);
68      }
69
70      private JLabel rateLabel;
71      private JTextField rateField;
72      private JButton button;
73      private JLabel resultLabel;
74      private JPanel panel;
75      private BankAccount account;
76
77      private static final int FRAME_WIDTH = 500;
78      private static final int FRAME_HEIGHT = 200;
79
80      private static final double DEFAULT_RATE = 5;
81      private static final double INITIAL_BALANCE = 1000;
82   }
```

Autoverificação da aprendizagem

20. O que acontece se você omitir o primeiro objeto `JLabel`?

21. Se um campo de texto contiver um inteiro, qual expressão você utilizará para ler o conteúdo?

10.11 Áreas de texto

Utilize um `JTextArea` para exibir múltiplas linhas de texto.

Na Seção 10.10, vimos como construir campos de texto. Um campo de texto contém uma única linha de texto. Para exibir múltiplas linhas de texto, utilize a classe `JTextArea`.

Ao construir uma área de texto, você pode especificar o número de linhas e de colunas:

```
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

Utilize o método `setText` para configurar o texto de um campo ou de uma área de texto. O método `append` adiciona texto ao fim de uma área de texto. Use caracteres de nova linha para separar linhas, assim:

```
textArea.append(account.getBalance() + "\n");
```

Se quiser utilizar um campo de texto ou uma área de texto apenas para fins de exibição, chame o método `setEditable` da seguinte maneira

```
textArea.setEditable(false);
```

Agora o usuário não pode mais editar o conteúdo do campo, mas seu programa ainda pode chamar `setText` e `append` para alterá-lo.

Como mostrado na Figura 4, as classes `JTextField` e `JTextArea` são subclasses da classe `JTextComponent`. Os métodos `setText` e `setEditable` são definidos na classe `JTextComponent` e herdados por `JTextField` e `JTextArea`. Mas o método `append` é definido na classe `JTextArea`.

Você pode adicionar barras de rolagem a qualquer componente com um `JScrollPane`.

Para adicionar barras de rolagem a uma área de texto, utilize um `JScrollPane`, assim:

```
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Adicione então a barra de rolagem ao painel. A Figura 13 mostra o resultado.

O programa de exemplo a seguir agrupa esses conceitos. Um usuário pode inserir números no campo de texto da taxa de juros e então clicar no botão "Add Interest"). A taxa de juros é aplicada e o saldo atualizado é acrescentado à área de texto. A área de texto tem barras de rolagem e não é editável.

Esse programa é semelhante ao programa visualizador de investimentos anterior, mas monitora todos os saldos bancários, não apenas o último.

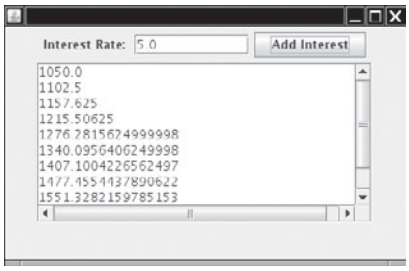


Figura 13
Aplicação de investimento com uma área de texto.

ch10/textarea/InvestmentFrame.java

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6  import javax.swing.JPanel;
7  import javax.swing.JScrollPane;
8  import javax.swing.JTextArea;
9  import javax.swing.JTextField;
10
11 /**
12    Um frame que mostra o crescimento de um investimento com juros variáveis.
13 */
14 public class InvestmentFrame extends JFrame
15 {
16    public InvestmentFrame()
17    {
18       account = new BankAccount(INITIAL_BALANCE);
19       resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
20       resultArea.setEditable(false);
21
22       // Usa métodos auxiliares
23       createTextField();
24       createButton();
25       createPanel();
26
27       setSize(FRAME_WIDTH, FRAME_HEIGHT);
28    }
29
30    private void createTextField()
31    {
32       rateLabel = new JLabel("Interest Rate: ");
33
34       final int FIELD_WIDTH = 10;
35       rateField = new JTextField(FIELD_WIDTH);
36       rateField.setText("" + DEFAULT_RATE);
37    }
38
39    private void createButton()
40    {
41       button = new JButton("Add Interest");
42
43       class AddInterestListener implements ActionListener
44       {
45          public void actionPerformed(ActionEvent event)
46          {
47             double rate = Double.parseDouble(
48                   rateField.getText());
49             double interest = account.getBalance()
50                   * rate / 100;
51             account.deposit(interest);
52             resultArea.append(account.getBalance() + "\n");
53          }
54       }
```

```
55
56         ActionListener listener = new AddInterestListener();
57         button.addActionListener(listener);
58      }
59
60      private void createPanel()
61      {
62         panel = new JPanel();
63         panel.add(rateLabel);
64         panel.add(rateField);
65         panel.add(button);
66         JScrollPane scrollPane = new JScrollPane(resultArea);
67         panel.add(scrollPane);
68         add(panel);
69      }
70
71      private JLabel rateLabel;
72      private JTextField rateField;
73      private JButton button;
74      private JTextArea resultArea;
75      private JPanel panel;
76      private BankAccount account;
77
78      private static final int FRAME_WIDTH = 400;
79      private static final int FRAME_HEIGHT = 250;
80
81      private static final int AREA_ROWS = 10;
82      private static final int AREA_COLUMNS = 30;
83
84      private static final double DEFAULT_RATE = 5;
85      private static final double INITIAL_BALANCE = 1000;
86   }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

22. Qual é a diferença entre um campo de texto e uma área de texto?

23. Por que o programa `InvestmentFrame` chama `resultArea.setEditable(false)`?

24. Como você modificaria o programa `InvestmentFrame` se não quisesse utilizar barras de rolagem?

COMO FAZER 10.1

Implementando uma interface gráfica com o usuário (*graphical user interface* – GUI)

Um programa GUI permite que usuários forneçam entradas e especifiquem ações. O programa `InvestmentViewer3` tem apenas uma entrada e uma ação. Programas mais sofisticados têm interações mais interessantes com o usuário, mas os princípios básicos são os mesmos.

Passo 1 Enumere as ações que seu programa precisa executar.

Por exemplo, o visualizador de investimento tem uma única ação, adicionar juros. Outros programas podem ter ações diferentes, talvez para fazer depósitos, inserir moedas e assim por diante.

Passo 2 Para cada ação, enumere as entradas que você precisa.

Por exemplo, o visualizador de investimentos tem uma única entrada: a taxa de juros. Outros programas podem ter entradas diferentes, como quantias em dinheiro, quantidades de produtos e assim por diante.

Passo 3 Para cada ação, enumere as saídas que você precisa mostrar.

O visualizador de investimentos tem uma única saída: o saldo atual. Outros programas podem mostrar diferentes quantidades, mensagens e assim por diante.

Passo 4 Forneça os componentes da interface com o usuário.

Nesse momento, você precisa utilizar botões para ações, campos de texto para entradas e rótulos para saídas. No Capítulo 18, veremos vários outros componentes de interface com o usuário que podem ser utilizados para ações e entradas. No Capítulo 3, você aprendeu a implementar seus próprios componentes para gerar saída gráfica, como gráficos ou desenhos.

Adicione os botões, campos de texto e outros componentes necessários a um frame. Neste capítulo, vimos como organizar interfaces com o usuário muito simples, adicionar todos os componentes a um único painel e adicionar esse painel ao frame. O Capítulo 18 mostra como você pode alcançar leiautes mais complexos.

Passo 5 Forneça as classes com as rotinas de tratamento de eventos.

Para cada botão, você precisa adicionar um objeto de uma classe ouvinte. As classes ouvintes devem implementar a interface `ActionListener`. Forneça uma classe para cada ação (ou um grupo de ações relacionadas) e coloque as instruções para a ação no método `actionPerformed`.

```
class Button1Listener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // a ação de button1 entra aqui
      . . .
   }
}
```

Lembre-se de declarar qualquer variável local acessada pelos métodos ouvintes como `final`.

Passo 6 Crie objetos ouvinte e anexe-os às fontes de evento.

Para eventos de ação, a fonte do evento é um botão ou um outro componente da interface com o usuário, ou um timer. Você precisa adicionar um objeto ouvinte a cada fonte de evento, assim:

```
ActionListener listener1 = new Button1Listener();
button1.addActionListener(listener1);
```

ERRO COMUM 10.8

Por padrão, componentes têm largura e altura zero

Os programas GUI de exemplo neste capítulo exibem os resultados em um rótulo ou em uma área de texto. Às vezes, você quer utilizar um componente gráfico, por exemplo, um gráfico ou diagrama. Você adiciona o componente gráfico ao painel:

```
panel.add(textField);
panel.add(button);
panel.add(chartComponent);
```

Mas o tamanho padrão para um componente é 0 por 0 pixels e o componente gráfico não estará visível. O correto é chamar o método `setPreferredSize`, da seguinte maneira:

```
chartComponent.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
```

Componentes GUI, como botões e campos de texto, sabem como calcular o seu tamanho preferido, mas você mesmo deve configurar o tamanho preferido dos componentes em que desenha.

DICA DE PRODUTIVIDADE 10.2

Reutilização de código

Suponha que você tenha a tarefa de escrever um outro programa de interface gráfica com o usuário que lê a entrada a partir de alguns campos de texto e exibe o resultado de alguns cálculos em um rótulo ou área de texto. Você não é obrigado a iniciar a partir do zero. Em vez disso, você pode – e freqüentemente deve – *reutilizar* o esboço de um programa existente, como a classe `InvestmentFrame` anterior.

Para reutilizar o código do programa, simplesmente faça uma cópia de um arquivo de programa e atribua à cópia um novo nome. Por exemplo, talvez você queira copiar `InvestmentFrame.java` para um arquivo `TaxReturnFrame.java`. Então remova o código que é claramente específico ao problema antigo, mas, no lugar, deixe o esboço. Isto é, mantenha o painel, o campo de texto, ouvintes de eventos e assim por diante. Preencha o código para seus novos cálculos. Por fim, atribua um novo nome aos títulos da classe, botão, frames, etc.

Depois de entender os princípios por trás dos ouvintes de eventos, frames e painéis, não há necessidade de repensá-los todas as vezes. Reutilizar a estrutura de um programa em funcionamento torna seu trabalho mais eficiente.

Mas a reutilização "copiando e renomeando" ainda é uma abordagem mecânica um pouco propensa a erros. Continua sendo melhor empacotar estruturas de programa reutilizáveis em um conjunto de classes comuns. O mecanismo de herança permite projetar classes para reutilização sem copiar e colar.

RESUMO DO CAPÍTULO

1. A herança é um mecanismo para estender as classes existentes adicionando métodos e campos.
2. A classe mais geral chama-se superclasse. A classe mais especializada que herda da superclasse chama-se subclasse.
3. Cada classe estende a classe `Object`, direta ou indiretamente.
4. Herdar a herança de uma classe é diferente de implementar uma interface: a subclasse herda o comportamento e o estado da superclasse.
5. Uma vantagem da herança é a reutilização de código.
6. Ao definir uma subclasse, você especifica campos de instância adicionais, métodos adicionais e métodos modificados ou sobrescritos.
7. Conjuntos de classes podem formar hierarquias complexas de herança.
8. Uma subclasse não tem acesso aos campos privados de sua superclasse.
9. Utilize a palavra-chave `super` para chamar um método da superclasse.
10. Para chamar o construtor da superclasse, utilize a palavra-chave `super` na primeira instrução do construtor da subclasse.
11. Referências da subclasse podem ser convertidas em referências da superclasse.
12. O operador `instanceof` testa se um objeto é de um tipo específico.
13. Um campo ou método que não é declarado como `public`, `private` ou `protected` pode ser acessado por todas as classes no mesmo pacote, o que normalmente não é desejável.
14. Defina o método `toString` para que produza uma string que descreve o estado do objeto.
15. Defina o método `equals` para testar se dois objetos têm um estado igual.
16. O método `clone` cria um novo objeto com o mesmo estado de um objeto existente.
17. Defina uma subclasse `JFrame` para um frame complexo.
18. Utilize componentes `JTextField` para fornecer espaço para entrada de usuário. Posicione um `JLabel` ao lado de cada campo de texto.
19. Utilize um `JTextArea` para exibir múltiplas linhas de texto.
20. Você pode adicionar barras de rolagem a qualquer componente com um `JScrollPane`.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.awt.Component
   setPreferredSize
java.awt.Dimension
java.lang.Cloneable
java.lang.CloneNotSupportedException
java.lang.Object
   clone
   toString
javax.swing.JTextArea
   append
javax.swing.JTextField
javax.swing.text.JTextComponent
   getText
   isEditable
   setEditable
   setText
```

EXERCÍCIOS DE REVISÃO

★ **Exercício R10.1.** Qual é o saldo de `b` depois das seguintes operações?

```
SavingsAccount b = new SavingsAccount(10);
b.deposit(5000);
b.withdraw(b.getBalance() / 2);
b.addInterest();
```

★ **Exercício R10.2.** Descreva todos os construtores da classe `SavingsAccount`. Liste todos os métodos que são herdados da classe `BankAccount`. Liste todos os métodos que são adicionados à classe `SavingsAccount`.

★★ **Exercício R10.3.** Você pode converter uma referência a superclasse em uma referência a subclasse? Uma referência a subclasse em uma referência a superclasse? Se sim, dê exemplos. Se não, explique por quê.

★★ **Exercício R10.4.** Identifique a superclasse e a subclasse em cada um dos seguintes pares de classes.

a. `Employee, Manager`

b. `Polygon, Triangle`

c. `GraduateStudent, Student`

d. `Person, Student`

e. `Employee, GraduateStudent`

f. `BankAccount, CheckingAccount`

g. `Vehicle, Car`

h. `Vehicle, Minivan`

i. `Car, Minivan`

j. `Truck, Vehicle`

★ **Exercício R10.5.** Suponha que a classe `Sub` estenda a classe `Sandwich`. Quais atribuições a seguir são válidas?

```
Sandwich x = new Sandwich();
Sub y = new Sub();
```

a. `x = y;`

b. `y = x;`

c. `y = new Sandwich();`

d. `x = new Sub();`

★ **Exercício R10.6.** Desenhe um diagrama de herança que mostre os relacionamentos de herança entre as classes:

- `Person`
- `Employee`
- `Student`
- `Instructor`
- `Classroom`
- `Object`

★★ **Exercício R10.7.** Em um sistema de simulação de tráfego orientado a objetos, temos as seguintes classes:

- `Vehicle`
- `Car`
- `Truck`
- `Sedan`
- `Coupe`
- `PickupTruck`
- `SportUtilityVehicle`
- `Minivan`
- `Bicycle`
- `Motorcycle`

Desenhe um diagrama de herança que mostre os relacionamentos entre essas classes.

★★ **Exercício R10.8.** Quais relacionamentos de herança você estabeleceria entre estas classes?

- `Student`
- `Professor`
- `TeachingAssistant`
- `Employee`
- `Secretary`
- `DepartmentChair`
- `Janitor`
- `SeminarSpeaker`
- `Person`
- `Course`
- `Seminar`
- `Lecture`
- `ComputerLab`

★★★ **Exercício R10.9.** Quais destas condições retornam `true`? Verifique na documentação Java os padrões de herança.

a. `Rectangle r = new Rectangle(5, 10, 20, 30);`

b. `if (r instanceof Rectangle) . . .`

c. `if (r instanceof Point) . . .`

d. `if (r instanceof Rectangle2D.Double) . . .`

e. `if (r instanceof RectangularShape) . . .`

f. `if (r instanceof Object) . . .`

g. `if (r instanceof Shape) . . .`

★★ **Exercício R10.10.** Explique os dois significados da palavra-chave `super`. Explique os dois significados da palavra-chave `this`. Como elas estão relacionadas?

★★★ **Exercício R10.11.** (Ardiloso.) Considere as duas chamadas

```
public class D extends B
{
   public void f()
   {
      this.g(); // 1
   }
   public void g()
   {
      super.g(); // 2
   }
   . . .
}
```

Qual delas é um exemplo de polimorfismo?

★★★ **Exercício R10.12.** Considere este programa:

```
public class AccountPrinter
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings
            = new SavingsAccount(0.5);

      CheckingAccount harrysChecking
            = new CheckingAccount(0);

      . . .
      endOfMonth(momsSavings);
      endOfMonth(harrysChecking);
      printBalance(momsSavings);
      printBalance(harrysChecking);
   }

   public static void endOfMonth(SavingsAccount savings)
   {
      savings.addInterest();
```

```
      }
      public static void endOfMonth(CheckingAccount checking)
      {
         checking.deductFees();
      }
      public static void printBalance(BankAccount account)
      {
         System.out.println("The balance is $"
               + account.getBalance());
      }
   }
```

As chamadas ao método `endOfMonths` são resolvidas por vinculação precoce ou por vinculação tardia? Dentro do método `printBalance`, a chamada a `getBalance` é resolvida por vinculação precoce ou por vinculação tardia?

★ **Exercício R10.13.** Explique os termos *cópia superficial* e *cópia profunda*.

★ **Exercício R10.14.** Qual atributo de acesso os campos de instância devem ter? Qual atributo de acesso os campos estáticos devem ter? E quanto aos campos `static` e `final`?

★ **Exercício R10.15.** Qual atributo de acesso os métodos de instância devem ter? O mesmo é verdadeiro para os métodos estáticos?

★★ **Exercício R10.16.** Os campos `System.in` e `System.out` são campos públicos estáticos. É possível sobrescrevê-los? Se sim, como?

★★ **Exercício R10.17.** Por que os campos públicos são perigosos? Os campos estáticos públicos são mais perigosos do que os campos de instância públicos?

★G **Exercício R10.18.** Qual é a diferença entre um rótulo, um campo de texto e uma área de texto?

★★G **Exercício R10.19.** Cite um método que é definido em `JTextArea`, um método que `JTextArea` herda de `JTextComponent` e um método que `JTextArea` herda de `JComponent`.

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P10.1.** Aperfeiçoe o método `addInterest` da classe `SavingsAccount` para calcular os juros sobre o saldo *mínimo* a partir da última chamada a `addInterest`. *Dica:* você também precisa modificar o método `withdraw` e adicionar um campo de instância para lembrar do saldo mínimo.

★★ **Exercício P10.2.** Adicione uma classe `TimeDepositAccount` (depósito a prazo fixo) à hierarquia das contas bancárias. Essa conta é igual a uma conta poupança, mas você promete deixar o dinheiro na conta durante um número predeterminado de meses e há uma penalidade para retiradas antecipadas. Construa essa conta com a taxa de juros e o número de meses do prazo predeterminado. No método `addInterest`, decremente a

contagem de meses. Se a contagem for positiva durante uma retirada, cobre a penalidade por essa retirada.

★ **Exercício P10.3.** Implemente uma subclasse `Square` que estende a classe `Rectangle`. No construtor, aceite as posições *x* e *y* do *centro* e o comprimento lateral do quadrado. Chame os métodos `setLocation` e `setSize` da classe `Rectangle`. Pesquise esses métodos na documentação da classe `Rectangle`. Também forneça um método `getArea` que calcula e retorna a área do quadrado. Escreva um programa de exemplo que solicita o centro e o comprimento lateral e então imprime o quadrado (utilizando o método `toString` que você herda de `Rectangle`) e a área do quadrado.

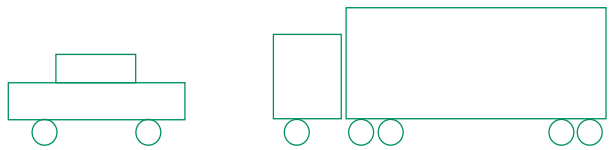
★ **Exercício P10.4.** Implemente uma superclasse `Person`. Crie duas classes, `Student` e `Instructor`, que herdam de `Person`. Uma pessoa tem um nome e um ano de nascimento. Um estudante tem uma especialização e um professor tem um salário. Escreva as definições de classe, os construtores e os métodos `toString` para todas as classes. Forneça um programa de teste que testa essas classes e métodos.

★★ **Exercício P10.5.** Crie uma classe `Employee` com um nome e salário. Faça uma classe `Manager` herdar de `Employee`. Adicione um campo de instância, chamado `department`, do tipo `String`. Forneça um método `toString` que imprime o nome do gerente, departamento e salário. Faça uma classe `Executive` herdar de `Manager`. Forneça os métodos `toString` apropriados para todas as classes. Forneça um programa de teste que testa essas classes e métodos.

★ **Exercício P10.6.** Escreva uma superclasse `Worker` e subclasses `HourlyWorker` e `SalariedWorker`. Cada trabalhador tem um nome e um percentual salarial. Escreva um método `computePay(int hours)` que calcula o pagamento semanal para cada trabalhador. Um horista é pago de acordo com o número real de horas trabalhadas, se `hours` for no máximo 40. Se o horista trabalhar mais de 40 horas, as horas excedentes serão com 50% de acréscimo. O funcionário assalariado é pago por 40 horas semanais, independentemente do número real de horas trabalhadas. Forneça um programa de teste que use polimorfismo para testar essas classes e métodos.

★★★ **Exercício P10.7.** Reorganize as classes de contas bancárias da seguinte maneira. Na classe `BankAccount`, introduza um método abstrato `endOfMonth` sem nenhuma implementação. Renomeie os métodos `addInterest` e `deductFees` para `endOfMonth` nas subclasses. Quais classes são abstratas agora e quais são concretas? Escreva um método estático `void test(BankAccount account)` que realiza cinco transações e então chama `endOfMonth`. Teste-o com instâncias de todas as classes de contas concretas.

★★★G **Exercício P10.8.** Implemente uma classe abstrata `Vehicle` e subclasses concretas `Car` e `Truck`. Um veículo tem uma posição na tela. Escreva métodos `draw` que desenham carros e caminhões da seguinte maneira:



Carro Caminhão

Escreva então um método `randomVehicle` que gera aleatoriamente referências a `Vehicle`, com uma probabilidade igual para construir carros e caminhões, com posições aleatórias. Chame-o 10 vezes e desenhe todos eles.

★G **Exercício P10.9.** Escreva uma interface gráfica com o usuário para uma classe de contas bancárias. Forneça campos de texto e botões para depositar e sacar dinheiro e também exibir o saldo atual em um rótulo.

★G **Exercício P10.10.** Escreva uma interface gráfica com o usuário para uma classe `Earthquake`. Forneça um campo de texto e botão para inserir a intensidade do terremoto. Exiba a descrição do terremoto em um rótulo.

★G **Exercício P10.11.** Escreva uma interface gráfica com o usuário para uma classe `DataSet`. Forneça campos de texto e botões para adicionar valores de ponto flutuante e exiba o mínimo, o máximo e a média atuais em um rótulo.

★G **Exercício P10.12.** Escreva uma aplicação com três campos de texto rotulados, cada um para a quantia inicial de uma conta poupança, a taxa de juros anual e o número de anos. Adicione um botão "Calculate" e uma área de leitura de texto para exibir o resultado, a saber, o saldo da conta poupança depois do final de cada ano.

★★G **Exercício P10.13.** Na aplicação do Exercício P10.12, substitua a área de texto por um gráfico de barras que mostre o saldo depois do final de cada ano.

★★★G **Exercício P10.14.** Escreva um programa que contém um campo de texto, um botão "Add Value" e um componente que desenhe um gráfico de barras dos números que um usuário digitou no campo de texto.

★★G **Exercício P10.15.** Escreva um programa que solicite ao usuário um inteiro e então desenhe o número de retângulos que o usuário solicitou em posições aleatórias em um componente.

★G **Exercício P10.16.** Escreva um programa que pede para o usuário inserir as coordenadas x e y do centro e um raio. Quando o usuário clicar em um botão "Draw", desenhe um círculo com esse centro e raio em um componente.

★★G **Exercício P10.17.** Escreva um programa que permite ao usuário especificar um círculo digitando o raio em um campo de texto e então clicando no centro. Observe que você não precisa de um botão "Draw".

★★★G **Exercício P10.18.** Escreva um programa que permite ao usuário especificar um círculo com dois cliques do mouse, o primeiro no centro e o segundo em um ponto na periferia. *Dica:* na rotina de tratamento do clique do mouse, você deve monitorar se já recebeu o ponto do centro em um clique anterior do mouse.

★★★G **Exercício P10.19.** Escreva um programa que desenha o mostrador de um relógio com um horário que o usuário insere em dois campos de texto (um para as horas, outro para os minutos).

Dica: você precisa determinar os ângulos do ponteiro das horas e do ponteiro dos minutos. O ângulo do ponteiro dos minutos é fácil: o ponteiro dos minutos percorre 360 graus em 60 minutos. O ângulo do ponteiro das horas é mais difícil; ele percorre 360 graus em 12×60 *minutos*.

★★G **Exercício P10.20.** Escreva um programa que pede para o usuário inserir um inteiro n e então desenha uma grade n-por-n.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 10.1.** Sua tarefa é programar robôs com comportamentos variados. Os robôs tentam sair de um labirinto, como o seguinte:

```
* *******
*      * *
* ***** *
* * *    *
* * *** *
*   *    *
*** * * *
*     * *
******* *
```

Um robô tem uma posição e um método `void move(Maze m)` que modifica esta posição. Forneça uma superclasse comum `Robot` cujo método `move` não faz nada. Forneça subclasses `RandomRobot`, `RightHandRuleRobot` e `MemoryRobot`. Cada um desses robôs tem uma estratégia diferente para escapar. `RandomRobot` simplesmente faz movimentos aleatórios. `RightHandRuleRobot` move-se pelo labirinto de forma que sua mão direita sempre toque uma parede. `MemoryRobot` lembra de todas as posições que já ocupou e nunca volta a uma posição que ele sabe que não tem saída.

★★★ **Projeto 10.2.** Implemente os métodos `toString`, `equals` e `clone` para todas as subclasses da classe `BankAccount`, bem como a classe `Bank` do Capítulo 7. Escreva testes de unidade que verificam se seus métodos funcionam corretamente. Certifique-se de testar um `Bank` que contém objetos provenientes de uma combinação de classes de contas.

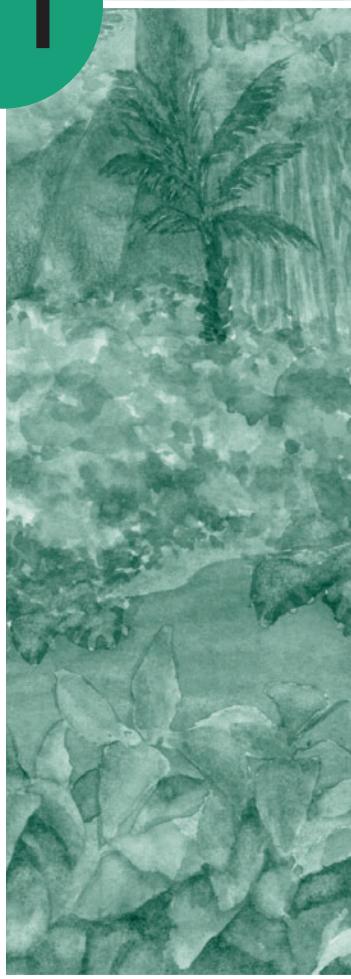
RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Dois campos de instância: `balance` e `interestRate`.
2. `deposit`, `withdraw`, `getBalance` e `addInterest`.
3. `Manager` é a subclasse; `Employee` é a superclasse.
4. Expressar o comportamento comum dos campos de texto e dos componentes de texto.
5. Precisamos de um contador que conte o número de retiradas e depósitos.
6. Ele precisa reduzir o saldo e não pode acessar o campo `balance` diretamente.
7. Para que a contagem possa refletir o número de transações para o mês seguinte.

8. Foi suficiente utilizar o construtor padrão da superclasse, que configura o saldo como zero.
9. Não – esse é um requisito somente dos construtores. Por exemplo, o método `CheckingAccount.deposit` primeiro incrementa a contagem das transações e então chama o método da superclasse.
10. Queremos utilizar o método para todos os tipos de contas bancárias. Se tivéssemos utilizado um parâmetro do tipo `SavingsAccount`, não poderíamos ter chamado o método com um objeto `CheckingAccount`.
11. Não podemos invocar o método `deposit` em uma variável do tipo `Object`.
12. O objeto é uma instância de `BankAccount` ou uma de suas subclasses.
13. O saldo de a permanece inalterado e a contagem de transações é incrementada duas vezes.
14. Esquecer acidentalmente do modificador `private`.
15. Qualquer método das classes no mesmo pacote.
16. Certamente deveria – a menos que, naturalmente, `x` seja `null`.
17. Se `toString` retornar uma string que descreve todos os campos de instância, você pode simplesmente chamar `toString` nos parâmetros implícitos e explícitos e comparar os resultados. Mas comparar os campos é mais eficiente do que convertê-los em strings.
18. Três: `InvestmentFrameViewer`, `InvestmentFrame` e `BankAccount`.
19. O construtor `InvestmentFrame` adiciona o painel a *ele próprio*.
20. Portanto, o campo de texto não é rotulado e o usuário não conhecerá seu propósito.
21. `Integer.parseInt(textField.getText())`
22. Um campo de texto contém uma única linha de texto; uma área de texto contém múltiplas linhas.
23. A área de texto é concebida para exibir a saída do programa. Ela não coleta a entrada de usuário.
24. Não construindo um `JScrollPane` e não adicionando o objeto `resultArea` diretamente ao frame.

Capítulo 11

Entrada/Saída e Tratamento de Exceção



Objetivos do capítulo

- Ler e gravar arquivos de texto
- Aprender a lançar exceções
- Projetar suas próprias classes de exceção
- Entender a diferença entre exceções verificadas e não-verificadas
- Aprender a capturar exceções
- Saber quando e onde capturar uma exceção

Este capítulo inicia com uma discussão sobre entrada e saída em arquivo. Sempre que você ler ou gravar dados, erros potenciais devem ser esperados. Um arquivo pode ter sido corrompido, excluído ou armazenado em outro computador que acabou de se desconectar da rede. Para lidar com essas questões, você precisa conhecer tratamento de exceções. O restante deste capítulo mostra como seus programas podem informar condições excepcionais e como podem se recuperar quando ocorrer uma condição excepcional.

Conteúdo do Capítulo

11.1 Lendo e gravando arquivos de texto 452
Erro comum 11.1: Barras invertidas em nomes de arquivo 454
Tópico avançado 11.1: Caixas de diálogo de arquivo ⊕
Tópico avançado 11.2: Argumentos de linha de comando ⊕

11.2 Lançando exceções 455
Sintaxe 11.1: Lançando uma exceção 457

11.3 Exceções verificadas e não-verificadas 458
Sintaxe 11.2: Especificação de exceção 460

11.4 Capturando exceções 460
Sintaxe 11.3: Bloco `try` geral 461
Dica de qualidade 11.1: Lance cedo, capture tarde 462
Dica de qualidade 11.2: Não silencie exceções 463

11.5 A cláusula `finally` 463
Sintaxe 11.4: Cláusula `finally` 464
Dica de qualidade 11.3: Não utilize `catch` e `finally` no mesmo comando `try` 465

11.6 Projetando seus próprios tipos de exceção 465
Dica de qualidade 11.4: Lance exceções específicas 466

11.7 Estudo de caso: um exemplo completo 467
Fato aleatório 11.1: O incidente com o foguete Ariane ⊕

11.1 Lendo e gravando arquivos de texto

Começamos este capítulo discutindo a tarefa comum de ler e gravar arquivos que contenham texto. Os exemplos são arquivos criados com um editor de textos simples, como o Bloco de Notas do Windows, bem como código-fonte Java e arquivos HTML.

O mecanismo mais simples para ler texto é utilizar a classe `Scanner`. Você já sabe utilizar a `Scanner` para ler entrada de console. Para ler a entrada a partir de um arquivo em disco, construa primeiramente um objeto `FileReader` com o nome do arquivo de entrada, então utilize o `FileReader` para construir um objeto `Scanner`:

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```

Ao ler arquivos de texto, utilize a classe `Scanner`.

Esse objeto `Scanner` lê texto do arquivo `input.txt`. Você pode utilizar o método `Scanner` (como `next`, `nextLine`, `nextInt` e `nextDouble`) para ler dados do arquivo de entrada.

Ao escrever arquivos de texto, utilize a classe `PrintWriter` e os métodos `print`/`println`.

Para escrever a saída em um arquivo, você constrói um objeto `PrintWriter` com o nome de arquivo dado, por exemplo

```
PrintWriter out = new PrintWriter("output.txt");
```

Se o arquivo de saída já existir, ele é esvaziado antes que os novos dados sejam gravados nele. Se não existir, um arquivo vazio é criado.

Utilize os métodos familiares `print` e `println` para enviar números, objetos e strings para um `PrintWriter`:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

Os métodos `print` e `println` convertem números em suas representações decimais de string e utilizam o método `toString` para converter objetos em strings.

Você deve fechar todos os arquivos quando tiver terminado de processá-los.

Quando terminar o processamento de um arquivo, confirme se *fechou* o `Scanner` ou `PrintWriter`:

```
in.close();
out.close();
```

Se seu programa sair sem fechar o `PrintWriter`, pode acontecer de nem toda a saída ser gravada no arquivo em disco.

O programa a seguir coloca esses conceitos em prática. Ele lê todas as linhas de um arquivo de entrada e as envia para o arquivo de saída, precedido por *números de linha*. Se o arquivo de entrada for:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

então o programa produz o seguinte arquivo de saída:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

Os números de linha são incluídos entre os delimitadores `/* */` a fim de que o programa possa ser utilizado para numerar arquivos-fonte em Java.

Essa é uma questão adicional que precisamos reexaminar. Quando o arquivo de entrada ou saída não existir, uma `FileNotFoundException` pode ocorrer. O compilador insiste que informemos o que o programa deve fazer quando isso acontece. (Nesse aspecto, a `FileNotFoundException` é diferente das exceções que você já encontrou. Discutiremos essa diferença em detalhes na Seção 11.3.) Em nosso programa de exemplo, pegamos a saída fácil e reconhecemos que o método `main` deve ser simplesmente terminado se a exceção ocorrer. Rotulamos o método `main` assim:

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

Você verá nas seções a seguir como lidar com exceções de maneira mais profissional.

ch11/fileio/LineNumberer.java

```
1  import java.io.FileReader;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  import java.util.Scanner;
5
6  public class LineNumberer
7  {
8     public static void main(String[] args)
9           throws FileNotFoundException
10    {
11       Scanner console = new Scanner(System.in);
12       System.out.print("Input file: ");
13       String inputFileName = console.next();
14       System.out.print("Output file: ");
```

```
15         String outputFileName = console.next();
16
17         FileReader reader = new FileReader(inputFileName);
18         Scanner in = new Scanner(reader);
19         PrintWriter out = new PrintWriter(outputFileName);
20         int lineNumber = 1;
21
22         while (in.hasNextLine())
23         {
24            String line = in.nextLine();
25            out.println("/* " + lineNumber + " */ " + line);
26            lineNumber++;
27         }
28
29         out.close();
30      }
31   }
```

Autoverificação da aprendizagem

1. O que acontece quando você fornece o mesmo nome para os arquivos de entrada e saída ao programa `LineNumberer`?
2. O que acontece quando você fornece o nome de um arquivo de entrada inexistente ao programa `LineNumberer`?

Erro comum 11.1

Barras invertidas em nomes de arquivo

Ao especificar um nome de arquivo como uma string constante e se o nome contiver caracteres do tipo barras invertidas (como em um nome de arquivo Windows), você deve duplicar cada barra invertida:

```
in = new FileReader("c:\\homework\\input.dat");
```

Lembre-se de que uma barra invertida única entre strings é um caractere de escape que é combinado com outro caractere para formar um significado especial, como `\n` para um caractere de nova linha. A combinação `\\` indica uma única barra invertida.

Quando um usuário fornecer um nome de arquivo para um programa, porém, o usuário não deve digitar as barras invertidas duas vezes.

Tópico avançado 11.1

Caixas de diálogo de arquivo

O Tópico Avançado 11.1 mostra como apresentar uma caixa de diálogo de escolha de arquivo para os usuários de seus programas.

TÓPICO AVANÇADO 11.2

Argumentos de linha de comando

O Tópico Avançado 11.2 mostra como processar *argumentos de linha de comando*, strings que são fornecidas depois que o nome de um programa é carregado a partir de um shell de comandos. Os argumentos de linha de comando são passados para o parâmetro `args` do método `main`.

11.2 Lançando exceções

Há dois aspectos principais no tratamento de exceções: *relato* e *recuperação*. Um desafio importante do tratamento de erro é que o ponto de relato normalmente está bem adiante do ponto de recuperação. Por exemplo, o método `get` da classe `ArrayList` pode detectar que um elemento inexistente está sendo acessado, mas não tem informações suficientes para decidir o que fazer sobre essa falha. O usuário deve ser solicitado a tentar uma operação diferente? O programa deve ser abortado depois de salvar o trabalho do usuário? A lógica para essas decisões está contida em uma parte diferente do código do programa.

Em Java, *o tratamento de exceções* fornece um mecanismo flexível para passar o controle do ponto de relato de erro para um módulo de tratamento de recuperação competente. No restante deste capítulo, examinaremos os detalhes desse mecanismo.

Para sinalizar uma condição excepcional, utilize a instrução `throw` para lançar um objeto de exceção.

Ao detectar uma condição de erro, seu trabalho é realmente fácil. Simplesmente use o `throw` para lançar um objeto de exceção apropriado e seu trabalho estará pronto. Por exemplo, suponha que alguém tente retirar dinheiro demais de uma conta bancária.

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
         // E agora?
      . . .
   }
   . . .
}
```

Primeiramente procure uma classe de exceção apropriada. A biblioteca Java fornece muitas classes para sinalizar todos os tipos de condições excepcionais. A Figura 1 mostra as mais úteis.

Procure por um tipo de exceção que possa descrever sua situação. E quanto a `IllegalStateException`? A conta bancária está em um estado inválido para a operação `withdraw`? Não realmente – algumas operações `withdraw` poderiam ser bem-sucedidas. E o valor de parâmetro é ilegal? Ele é de fato. É simplesmente grande demais. Portanto, vamos lançar uma `IllegalArgumentException`. (O termo argumento é um termo alternativo para um valor de parâmetro.)

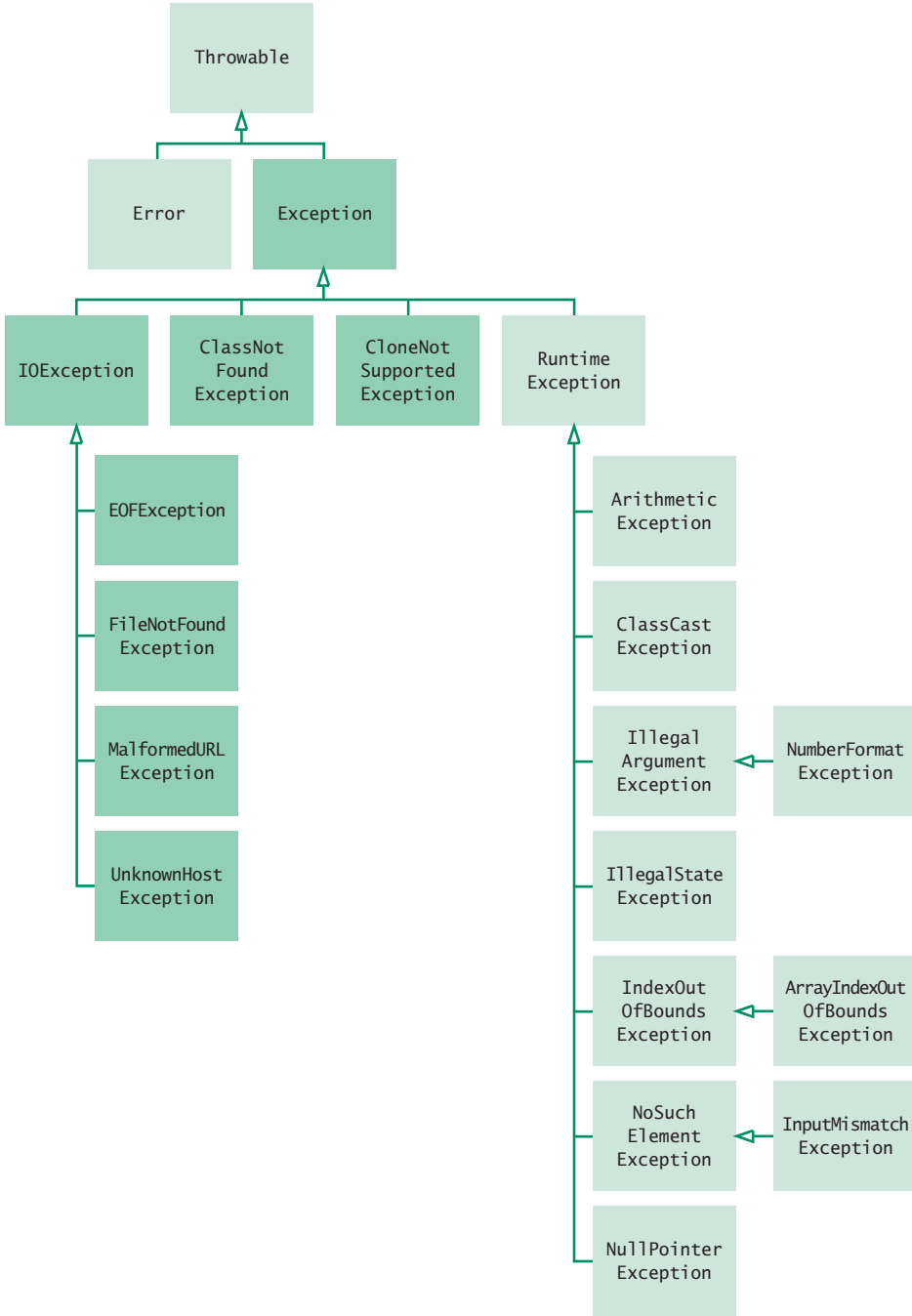


Figura 1 Hierarquia de classes de exceção.

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception
               = new IllegalArgumentException("Amount exceeds balance");
         throw exception;
      }
      balance = balance - amount;
   }
   . . .
}
```

Na verdade, você não precisa armazenar o objeto de exceção em uma variável. Você pode simplesmente lançar o objeto que o operador `new` retorna:

```
throw new IllegalArgumentException("Amount exceeds balance");
```

Ao lançar uma exceção, o método atual termina imediatamente.

Ao lançar uma exceção, a execução não continua com a próxima instrução, mas com um módulo de tratamento de exceções. Por enquanto, não nos preocuparemos com o tratamento de exceções. Esse é o tópico da Seção 11.4.

SINTAXE 11.1 Lançando uma exceção

`throw` *objetoDaExceção*`;`

Exemplo:

```
throw new IllegalArgumentException();
```

Objetivo:

Lançar uma exceção e transferir o controle para um módulo de tratamento (handler) desse tipo de exceção

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Como modificar o método `deposit` para assegurar que o saldo nunca fique negativo?
4. Suponha que você construa um novo objeto conta bancária com um saldo zero e, então, chame `withdraw(10)`. Qual é o valor de `balance` depois disso?

11.3 Exceções verificadas e não-verificadas

Há dois tipos de exceção: verificadas e não-verificadas. Exceções não-verificadas estendem a classe `RuntimeException` ou `Error`.

As exceções Java dividem-se em duas categorias, chamadas exceções *verificadas* e *não-verificadas*. Quando você chama um método que lança uma exceção verificada, o compilador verifica se você não a ignora. Você deve dizer ao compilador o que fará com a exceção se ela sempre for lançada. Por exemplo, todas as subclasses de `IOException` são exceções verificadas. Por outro lado, o compilador não exige que você monitore exceções não-verificadas. Exceções como `NumberFormatException`, `IllegalArgumentException` e `NullPointerException` são exceções não-verificadas. De modo geral, todas as exceções que pertencem às subclasses de `RuntimeException` são não-verificadas e todas as outras subclasses da classe `Exception` são verificadas. (Na Figura 1, as exceções verificadas são sombreadas em uma cor mais escura.) Há uma segunda categoria de erros internos que são informados lançando objetos do tipo `Error`. Um exemplo é o `OutOfMemoryError`, que é lançado quando toda memória disponível foi consumida. Esses são erros fatais que raramente acontecem e estão além do seu controle. Eles também são não-verificados.

Exceções verificadas ocorrem devido a circunstâncias externas que o programador não pode evitar. O compilador verifica se o programa trata essas exceções.

Por que ter dois tipos de exceção? Uma exceção verificada descreve um problema que possivelmente ocorrerá algumas vezes, independentemente de quão cuidadoso você seja. As exceções não-verificadas, por outro lado, são uma falha sua. Por exemplo, um fim de arquivo inesperado pode ser causado por forças além de seu controle, como um erro de disco ou uma conexão de rede interrompida. Mas você é o responsável pela `NullPointerException`, porque seu código estava errado quando ele tentava utilizar uma referência `null`.

O compilador não verifica se você trata ou não uma `NullPointer-Exception`, pois você deve testar suas referências quanto a `null` antes de utilizá-las, em vez de instalar um módulo de tratamento para essa exceção. O compilador realmente insiste que seu programa pode tratar condições de erro que você não pode impedir.

De fato, essas categorias não são perfeitas. Por exemplo, o método `Scanner.nextInt` lança uma `InputMismatchException` não-verificada se um usuário inserir uma entrada que não seja um inteiro. Uma exceção verificada teria sido mais apropriada porque o programador não pode impedir que os usuários insiram uma entrada incorreta. (Os projetistas da classe `Scanner` fizeram essa escolha para facilitar o uso dessa classe pelos programadores iniciantes.)

Como você pode ver na Figura 1, a maioria das exceções verificadas ocorre quando você lida com entrada e saída. Esse é um solo fértil para falhas externas fora de seu controle – um arquivo pode ter sido corrompido ou removido, uma conexão de rede pode estar sobrecarregada, um servidor pode ter falhado e assim por diante. Portanto, você precisará lidar com exceções verificadas principalmente ao programar com arquivos e fluxos.

Você viu como utilizar a classe `Scanner` para ler dados de um arquivo, passando um objeto `FileReader` para o construtor `Scanner`:

```
String filename = . . .;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

Entretanto, o construtor `FileReader` pode lançar uma `FileNotFoundException`. A `FileNotFoundException` é uma exceção verificada, então você precisa informar o compilador sobre o que você vai fazer a respeito. Há duas escolhas. Você pode tratar a exceção, utilizando as técnicas apresentadas na Seção 11.4. Ou pode simplesmente informar o compilador de que você está ciente dessa exceção e que quer que o seu método seja terminado quando ela ocorrer. O método que lê a entrada raramente sabe o que fazer com um erro inesperado, portanto, em geral, essa é a melhor opção.

Para declarar se um método deve ser terminado quando uma exceção verificada ocorre dentro dele, marque o método com um especificador `throws`.

```
public class DataSet
{
   public void read(String filename) throws FileNotFoundException
   {
      FileReader reader = new FileReader(filename);
      Scanner in = new Scanner(reader);
      . . .
   }
   . . .
}
```

Adicione um especificador `throws` a um método que possa lançar uma exceção verificada.

A cláusula `throws`, por sua vez, sinaliza o chamador de seu método que uma `FileNotFoundException` poderá acontecer. Então o chamador precisa tomar a mesma decisão – tratar a exceção ou dizer ao seu chamador que a exceção pode ser lançada.

Se seu método puder lançar exceções verificadas de diferentes tipos, você deve separar os nomes de classe de exceção com vírgulas:

```
public void read(String filename)
      throws IOException, ClassNotFoundException
```

Sempre tenha em mente que as classes de exceção formam uma hierarquia de herança. Por exemplo, `FileNotFoundException` é uma subclasse de `IOException`. Portanto, se um método pode lançar tanto uma `IOException` como uma `FileNotFoundException`, você só o marca como `throws IOException`.

De certo modo, parece irresponsável não tratar uma exceção quando se sabe que ela aconteceu. Porém, em geral, é melhor não capturar uma exceção se você não souber reparar a situação. Afinal de contas, o que você pode fazer com um método `read` de baixo nível? Você pode informar ao usuário? Como? Enviando uma mensagem para `System.out`? Você não sabe se esse método é chamado em um programa gráfico ou em um sistema embutido (como uma máquina de venda automática), onde o usuário nunca pode ver `System.out`. E mesmo se os usuários puderem ver a mensagem de erro, como saber se eles entendem inglês? Sua classe pode ser utilizada para construir uma aplicação para usuários de outro país. Se não puder dizer ao usuário, você poderá corrigir os dados e continuar? Como? Se você configurar uma variável como zero, `null` ou uma string vazia, isso pode simplesmente fazer com que o programa trave mais tarde, com um mistério muito maior.

Naturalmente, alguns métodos no programa sabem se comunicar com o usuário ou adotam outra ação corretiva. Permitindo que a exceção alcance esses métodos, você possibilita o processamento da exceção por um módulo de tratamento competente.

SINTAXE 11.2 Especificação de exceção

EspecificadorDeAcesso tipoDeRetorno nomeDoMétodo(tipoDeParâmetro nomeDoParâmetro, . . .)
`throws` *ClasseDaExceção, ClasseDaExceção, . . .*

Exemplo:

```
public void read(FileReader in)
      throws IOException
```

Objetivo:

Indicar as exceções verificadas que esse método pode lançar

AUTOVERIFICAÇÃO DA APRENDIZAGEM

5. Suponha que um método chame o construtor `FileReader` e o método `read` da classe `FileReader`, que pode lançar uma `IOException`. Qual especificação `throws` você deve utilizar?
6. Por que uma `NullPointerException` não é uma exceção verificada?

11.4 Capturando exceções

Em um método que está pronto para tratar um determinado tipo de exceção, coloque as instruções que podem criar a exceção dentro de um bloco `try` e o módulo de tratamento dentro de uma cláusula `catch`.

Cada exceção deve ser tratada em algum lugar de seu programa. Se uma exceção não tiver nenhum módulo de tratamento, uma mensagem de erro é impressa e seu programa termina. Isso pode ser bom para um programa de estudante. Mas você não gostaria que um programa escrito profissionalmente parasse de funcionar só porque algum método detectou um erro inesperado. Portanto, você deve instalar módulos de tratamento de exceção para todas as exceções que o programa possa lançar.

Você instala um módulo de tratamento de exceções com a instrução `try/catch`. Cada bloco `try` contém uma ou mais instruções que podem causar uma exceção. Cada cláusula `catch` contém o módulo de tratamento para um tipo de exceção. Eis um exemplo:

```
try
{
   String filename = . . .;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
```

```
   }
   catch (NumberFormatException exception)
   {
      System.out.println("Input was not a number");
   }
```

Três exceções podem ser lançadas nesse bloco `try`: o construtor `FileReader` pode lançar uma `FileNotFoundException`, `Scanner.next` pode lançar uma `NoSuchElementException` e `Integer.parseInt` pode lançar uma `NumberFormatException`.

SINTAXE 11.3 Bloco `try` geral

```
try
{
   instrução
   instrução
   . . .
}
catch (ClasseDaExceção objetoDaExceção)
{
   instrução
   instrução
   . . .
}
catch (ClasseDaExceção objetoDaExceção)
{
   instrução
   instrução
   . . .
}
. . .
```

Exemplo:

```
try
{
   System.out.println("How old are you?");
   int age = in.nextInt();
   System.out.println("Next year, you'll be " + (age + 1));
}
catch (InputMismatchException exception)
{
   exception.printStackTrace();
}
```

Objetivo:

Executar uma ou mais instruções que possam gerar exceções. Se ocorrer uma exceção e ela corresponder a uma das cláusulas `catch`, execute a primeira que corresponder. Se não ocorrer nenhuma, ou uma exceção que for lançada não corresponder a quaisquer cláusulas `catch` então pule essas cláusulas.

Se alguma dessas exceções for realmente lançada, então as demais instruções no bloco `try` serão puladas. Eis o que acontece com os vários tipos de exceção:

- Se uma `FileNotFoundException` é lançada, então a cláusula `catch` da `IOException` é executada. (Lembre-se de que `FileNotFoundException` é uma subclasse de `IOException`.)
- Se ocorrer uma `NumberFormatException`, então a segunda cláusula `catch` é executada.
- Uma `NoSuchElementException` *não é capturada* por nenhuma das cláusulas `catch`. A exceção permanece lançada até que ela seja capturada por outro bloco `try`.

Quando o bloco `catch (IOException exception)` é executado, então é porque algum método no bloco `try` falhou com uma `IOException`. A variável `exception` contém uma referência ao objeto de exceção que foi lançado. A cláusula `catch` pode analisar esse objeto para descobrir mais detalhes sobre a falha. Por exemplo, você pode obter uma impressão da cadeia de chamadas do método que levou à exceção, chamando

```
exception.printStackTrace()
```

Nessas cláusulas `catch` de exemplo, apenas informamos o usuário sobre a origem do problema. Uma maneira melhor de lidar com a exceção seria dar outra chance para o usuário fornecer uma entrada correta – veja a Seção 11.7 para uma solução.

É importante lembrar-se de que você deve colocar as cláusulas `catch` somente em métodos onde você possa tratar competentemente o tipo específico de exceção.

Autoverificação da aprendizagem

7. Suponha que o arquivo com o nome de arquivo dado exista e não tenha conteúdo. Rastreie o fluxo de execução no bloco `try` nessa seção.
8. Há diferença entre capturar exceções verificadas e não-verificadas?

Dica de qualidade 11.1

Lance cedo, capture tarde

É melhor declarar que um método lança uma exceção verificada do que tratá-la precariamente.

Quando um método notar um problema que não pode resolver, em geral, é melhor lançar uma exceção em vez de tentar fazer um reparo malfeito (como não fazer nada ou retornar um valor padrão).

Inversamente, um método só deve capturar uma exceção se realmente puder reparar a situação. Caso contrário, a melhor solução é simplesmente deixar a exceção se propagar para seu chamador, permitindo que ela seja capturada por um módulo de tratamento competente.

Esses princípios podem ser resumidos com o lema "lançar cedo, capturar tarde".

DICA DE QUALIDADE 11.2

Não silencie exceções

Quando você chama um método que lança uma exceção verificada sem especificar um módulo de tratamento, o compilador reclama. Com o seu entusiasmo de continuar o trabalho, é compreensível um impulso para calar o compilador silenciando a exceção:

```
try
{
   FileReader reader = new FileReader(filename);
   // O compilador reclamou de FileNotFoundException
   . . .
}
catch (Exception e) {} // E agora, vai reclamar do quê?!
```

A rotina de tratamento de exceção vazia engana o compilador fazendo-o pensar que a exceção foi tratada. Com o tempo, vê-se claramente que isso é uma má idéia. As exceções foram projetadas para informar problemas a um módulo de tratamento competente. Instalar um módulo de tratamento incompetente simplesmente oculta uma condição de erro que pode ser grave.

11.5 A cláusula finally

Ocasionalmente, você precisará adotar alguma ação se uma exceção for lançada. A construção `finally` é utilizada para tratar essa situação. Eis uma situação típica.

É importante fechar um `PrintWriter` para assegurar que toda saída seja gravada no arquivo. No segmento de código a seguir, abrimos um gravador, chamamos um ou mais métodos e então o fechamos:

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // Talvez nunca chegue aqui
```

Agora suponha que um dos métodos antes da última linha lance uma exceção. Então a chamada a `close` nunca é executada! Resolva esse problema colocando a chamada a `close` dentro de uma cláusula `finally`:

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

Depois que se entra em um bloco `try`, as instruções em uma cláusula `finally` devem ser garantidamente executadas, quer uma exceção seja lançada ou não.

Em um caso normal, não há problema. Quando o bloco `try` é concluído, a cláusula `finally` é executada e o gravador é fechado. Entretanto, se ocorrer uma exceção, a cláusula `finally` também é executada antes de a exceção ser passada para seu módulo de tratamento.

Utilize a cláusula `finally` sempre que precisar fazer alguma limpeza, como fechar um arquivo, para assegurar que a limpeza aconteça independentemente de como o método termina.

Também é possível ter uma cláusula `finally` após uma ou mais cláusulas `catch`. Então o código na cláusula `finally` é executado sempre que o bloco `try` sair de uma destas três maneiras:

1. Depois de completar a última instrução do bloco `try`
2. Depois de completar a última instrução de uma cláusula `catch`, se esse bloco `try` capturou uma exceção
3. Quando uma exceção é lançada no bloco `try` e não é capturada

SINTAXE 11.4 Cláusula `finally`

```
try
{
   instrução
   instrução
   . . .
}
finally
{
   instrução
   instrução
   . . .
}
```

Exemplo:

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

Objetivo:

Assegurar que as instruções na cláusula `finally` são executadas quer as instruções no bloco `try` lancem uma exceção ou não

Entretanto, recomendamos não misturar as cláusulas `catch` e `finally` no mesmo bloco `try` – veja a Dica de Qualidade 11.3.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

9. Por que a variável `out` foi declarada fora do bloco `try`?
10. Suponha que o arquivo com o nome dado não exista. Rastreie o fluxo de execução do segmento de código nessa seção.

DICA DE QUALIDADE 11.3

Não utilize `catch` e `finally` no mesmo comando `try`

É tentador combinar as cláusulas `catch` e `finally`, mas o código resultante pode ser difícil de entender. Em vez disso, você deve utilizar um comando `try/finally` para fechar recursos e um comando `try/catch` separado para tratar erros. Por exemplo,

```
try
{
   PrintWriter out = new PrintWriter(filename);
   try
   {
      // Grava a saída
   }
   finally
   {
      out.close();
   }
}
catch (IOException exception)
{
   // Trata a exceção
}
```

Observe que as instruções aninhadas funcionam corretamente se a chamada `out.close()` lançar uma exceção – veja o Exercício R11.18.

11.6 Projetando seus próprios tipos de exceção

Às vezes nenhum dos tipos de exceção padrão descreve muito bem sua condição de erro específica. Nesse caso, você pode projetar sua própria classe de exceções. Considere uma conta bancária. Vamos informar uma `InsufficientFundsException` quando for feita uma tentativa de sacar de uma conta bancária um valor que excede o saldo atual.

```
if (amount > balance)
{
   throw new InsufficientFundsException(
         "withdrawal of " + amount + " exceeds balance of " + balance);
}
```

Você pode projetar seus próprios tipos de exceção – as subclasses de `Exception` ou `RuntimeException`.

Agora é necessário definir a classe `InsufficientFundsException`. Essa deve ser uma exceção verificada ou não-verificada? É culpa de algum evento externo ou do programador? O nosso ponto de vista é de que o programador poderia ter evitado a condição excepcional – afinal de contas, seria uma questão fácil verificar se `amount <= account.getBalance()` antes de chamar o método `withdraw`. Portanto, a exceção deve ser uma exceção não-verificada e estender a classe `RuntimeException` ou uma de suas subclasses.

É habitual fornecer dois construtores para uma classe de exceções: um construtor padrão e um construtor que aceita uma mensagem string para descrever a causa da exceção. Eis a definição da classe de exceções.

```
public class InsufficientFundsException
      extends RuntimeException
{
   public InsufficientFundsException() {}

   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```

Quando a exceção é capturada, a string da mensagem pode ser recuperada utilizando o método `getMessage` da classe `RunTimeException`.

Autoverificação da aprendizagem

11. Qual é o propósito da chamar `super(message)` no segundo construtor de `InsufficientFundsException`?
12. Suponha que você leia dados de conta bancária a partir de um arquivo. Contra sua expectativa, o próximo valor de entrada não é do tipo `double`. Você decide implementar uma `BadDataException`. Qual classe de exceção você deve estender?

Dica de qualidade 11.4

Lance exceções específicas

Ao lançar uma exceção, você deve escolher uma classe de exceção que descreva a situação com a maior exatidão possível. Por exemplo, seria uma má idéia simplesmente lançar um objeto `RuntimeException` quando uma conta bancária apresentar saldo insuficiente. Sem dúvida, isso dificultaria muito a tarefa de capturar a exceção. Afinal de contas, se você capturou todas as exceções do tipo `RuntimeException`, sua cláusula `catch` também seria ativada por exceções do tipo `NullPointerException`, `ArrayIndexOutOfBoundsException` e assim por diante. Você então precisaria examinar cuidadosamente o objeto de exceção e tentar deduzir se a exceção foi causada por insuficiência de fundos.

Se a biblioteca padrão não tiver uma classe de exceções que descreva sua situação de erro específica, simplesmente defina uma nova classe de exceção.

11.7 Estudo de caso: um exemplo completo

Esta seção mostra passo a passo um exemplo completo de um programa com tratamento de exceções. O programa solicita a um usuário o nome de um arquivo. Espera-se que o arquivo contenha os valores de dados. A primeira linha do arquivo contém o número total de valores e as linhas restantes contêm os dados. Um arquivo típico de entrada é parecido com este:

```
3
1.45
-2.1
0.05
```

O que pode dar errado? Há dois riscos principais.

- O arquivo pode não existir.
- Os dados do arquivo podem estar no formato errado.

Você pode detectar essas falhas? O construtor `FileReader` lançará uma exceção quando o arquivo não existir. Os métodos que processam os valores de entrada precisam lançar uma exceção quando localizarem um erro na formatação dos dados.

Que exceções podem ser lançadas? O construtor `FileReader` lança uma `FileNotFoundException` quando o arquivo não existir, o que é muito apropriado em nossa situação. O método `close` da classe `FileReader` pode lançar uma `IOException`. Por fim, quando os dados de arquivo estiverem com a formatação errada, lançaremos uma `BadDataException`, uma classe personalizada de exceções verificadas. Utilizamos uma exceção verificada porque corrupção em um arquivo de dados está além do controle do programador.

É possível reparar as falhas que as exceções informam? Somente o método `main` do programa `DataAnalyzer` interage com o usuário. Ele captura as exceções, imprime mensagens de erro relevantes e dá outra chance para o usuário inserir um arquivo correto.

ch11/data/DataAnalyzer.java

```
1  import java.io.FileNotFoundException;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  /**
6     Esse programa lê um arquivo contendo números e analisa seu conteúdo.
7     Se o arquivo não existir ou contiver strings que não são números, uma
8     mensagem de erro é exibida.
9  */
10 public class DataAnalyzer
11 {
12    public static void main(String[] args)
13    {
```

```
14         Scanner in = new Scanner(System.in);
15         DataSetReader reader = new DataSetReader();
16
17         boolean done = false;
18         while (!done)
19         {
20            try
21            {
22               System.out.println("Please enter the file name: ");
23               String filename = in.next();
24
25               double[] data = reader.readFile(filename);
26               double sum = 0;
27               for (double d : data) sum = sum + d;
28               System.out.println("The sum is " + sum);
29               done = true;
30            }
31            catch (FileNotFoundException exception)
32            {
33               System.out.println("File not found.");
34            }
35            catch (BadDataException exception)
36            {
37               System.out.println("Bad data: " + exception.getMessage());
38            }
39            catch (IOException exception)
40            {
41               exception.printStackTrace();
42            }
43         }
44      }
45   }
```

As duas primeiras cláusulas `catch` no método `main` fornecem um informe de erro legível por pessoas se o arquivo não for localizado ou se dados inválidos forem encontrados. Entretanto, se ocorrer outra `IOException`, então imprimimos o rastreamento de pilha para que um programador possa diagnosticar o problema.

O seguinte método `readFile` da classe `DataSetReader` constrói o objeto `Scanner` e chama o método `readData`. Ele é completamente indiferente em relação a qualquer exceção. Se houver um problema com o arquivo de entrada, ele simplesmente passa a exceção para seu chamador.

```
public double[] readFile(String filename)
      throws IOException, BadDataException
{
   FileReader reader = new FileReader(filename);
   try
   {
      Scanner in = new Scanner(reader);
      readData(in);
   }
   finally
   {
      reader.close();
```

```
      }
      return data;
   }
```

Note como a cláusula `finally` assegura que o arquivo é fechado mesmo quando ocorre uma exceção.

Note também que o especificador `throws` do método `readFile` não precisa incluir a classe `FileNotFoundException` porque é uma subclasse de `IOException`.

Em seguida, eis o método `readData` da classe `DataSetReader`. Ele lê o número de valores, constrói um array e chama `readValue` para cada valor de dados.

```
private void readData(Scanner in) throws BadDataException
{
   if (!in.hasNextInt())
      throw new BadDataException("Length expected");
   int numberOfValues = in.nextInt();
   data = new double[numberOfValues];

   for (int i = 0; i < numberOfValues; i++)
      readValue(in, i);

   if (in.hasNext())
      throw new BadDataException("End of file expected");
}
```

Esse método procura dois erros potenciais. Talvez o arquivo não inicie com um inteiro, ou tenha dados adicionais depois de ler todos os valores.

Entretanto, esse método não faz nenhuma tentativa de capturar todas as exceções. Além disso, se o método `readValue` lançar uma exceção – o que acontecerá se não houver valores suficientes no arquivo – a exceção simplesmente será passada para o chamador.

Eis o método `readValue`:

```
private void readValue(Scanner in, int i) throws BadDataException
{
   if (!in.hasNextDouble())
      throw new BadDataException("Data value expected");
   data[i] = in.nextDouble();
}
```

Para ver o tratamento de exceções em ação, examine um cenário de erro específico.

1. `DataAnalyzer.main` chama `DataSetReader.readFile`.
2. `readFile` chama `readData`.
3. `readData` chama `readValue`.
4. `readValue` não localiza o valor esperado e lança uma `BadDataException`.
5. `readValue` não tem nenhum módulo de tratamento para a exceção e termina imediatamente.
6. `readData` não tem nenhum módulo de tratamento para a exceção e termina imediatamente.
7. `readFile` não tem nenhum módulo de tratamento para a exceção e termina imediatamente depois de executar a cláusula `finally` e fechar o arquivo.
8. `DataAnalyzer.main` tem um módulo de tratamento para uma `BadDataException`. Esse módulo de tratamento imprime uma mensagem para o usuário. Depois, o usuário tem outra chance de inserir um nome de arquivo. Observe que as instruções que calculam a soma dos valores foram puladas.

Esse exemplo mostra a separação entre detecção do erro (no método `DataSetReader.readValue`) e tratamento do erro (no método `DataAnalyzer.main`). Entre os dois estão os métodos `readData` e `readFile`, que simplesmente passam as exceções para frente.

ch11/data/DataSetReader.java

```
1  import java.io.FileReader;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  /**
6     Lê um conjunto de dados de um arquivo. O arquivo deve ter o formato
7     numberOfValues
8     value1
9     value2
10    . . .
11 */
12 public class DataSetReader
13 {
14    /**
15       Lê um conjunto de dados.
16       @param filename nome do arquivo que armazena os dados
17       @return dados no arquivo
18    */
19    public double[] readFile(String filename)
20          throws IOException, BadDataException
21    {
22       FileReader reader = new FileReader(filename);
23       try
24       {
25          Scanner in = new Scanner(reader);
26          readData(in);
27       }
28       finally
29       {
30          reader.close();
31       }
32       return data;
33    }
34
35    /**
36       Lê todos os dados.
37       @param in scanner que varre os dados
38    */
39    private void readData(Scanner in) throws BadDataException
40    {
41       if (!in.hasNextInt())
42          throw new BadDataException("Length expected");
43       int numberOfValues = in.nextInt();
```

```
44         data = new double[numberOfValues];
45
46         for (int i = 0; i < numberOfValues; i++)
47            readValue(in, i);
48
49         if (in.hasNext())
50            throw new BadDataException("End of file expected");
51      }
52
53      /**
54         Lê um valor dos dados.
55         @param in scanner que varre os dados
56         @param i posição do valor a ler
57      */
58      private void readValue(Scanner in, int i) throws BadDataException
59      {
60         if (!in.hasNextDouble())
61            throw new BadDataException("Data value expected");
62         data[i] = in.nextDouble();
63      }
64
65      private double[] data;
66   }
```

ch11/data/BadDataException.java

```
1   /**
2      Essa classe informa dados de entrada inválidos.
3   */
4   public class BadDataException extends Exception
5   {
6      public BadDataException() {}
7      public BadDataException(String message)
8      {
9         super(message);
10     }
11  }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

13. Por que o método `DataSetReader.readFile` não captura nenhuma exceção?
14. Suponha que o usuário especifique um arquivo que existe e está vazio. Rastreie o fluxo de execução.

FATO ALEATÓRIO 11.1

O incidente com o foguete Ariane

O Fato Aleatório 11.1 conta a história do foguete Ariane 5 que se autodestruiu devido a uma exceção não-tratada.

RESUMO DO CAPÍTULO

1. Para ler arquivos de texto, utilize a classe `Scanner`.
2. Para gravar arquivos de texto, utilize a classe `PrintWriter` e os métodos `print`/`println`.
3. Você deve fechar todos os arquivos quando tiver terminado de processá-los.
4. Para sinalizar uma condição excepcional, utilize a instrução `throw` para lançar um objeto do tipo exceção.
5. Ao lançar uma exceção, o método atual termina imediatamente.
6. Há dois tipos de exceção: verificadas e não-verificadas. As exceções não-verificadas estendem a classe `RuntimeException` ou `Error`.
7. Exceções verificadas são lançadas devido a circunstâncias externas que o programador não pode evitar. O compilador verifica se o programa trata essas exceções.
8. Adicione um especificador `throws` a um método que possa lançar uma exceção verificada.
9. Em um método que está pronto para tratar um determinado tipo de exceção, coloque as instruções que podem produzir a exceção dentro de um bloco `try` e o módulo de tratamento dentro de uma cláusula `catch`.
10. É melhor declarar que um método lança uma exceção verificada do que tratar a exceção precariamente.
11. Depois que se entra em um bloco `try`, as instruções em uma cláusula `finally` devem ser garantidamente executadas, quer uma exceção seja lançada ou não.
12. Você pode projetar seus próprios tipos de exceção – subclasses de `Exception` ou `RuntimeException`.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.io.EOFException
java.io.File
   exists
java.io.FileNotFoundException
java.io.FileReader
java.io.IOException
java.io.PrintWriter
   close
   print
   println
java.lang.Error
java.lang.IllegalArgumentException
java.lang.IllegalStateException
java.lang.NullPointerException
java.lang.NumberFormatException
java.lang.RuntimeException
java.lang.Throwable
   getMessage
   printStackTrace
java.util.InputMismatchException
java.util.NoSuchElementException
java.util.Scanner
   close
javax.swing.JFileChooser
   getSelectedFile
   showOpenDialog
   showSaveDialog
```

EXERCÍCIOS DE REVISÃO

★★ **Exercício R11.1.** O que acontece se você tentar abrir um arquivo para leitura que não existe? O que acontece se você tenta abrir um arquivo para gravação que não existe?

★★★ **Exercício R11.2.** O que acontece se você tentar abrir um arquivo para gravação, mas o arquivo ou dispositivo for protegido contra gravação (às vezes chamado de *read-only*)? Experimente fazer isso usando um pequeno programa de teste.

★ **Exercício R11.3.** Como abrir um arquivo cujo nome contém barras invertidas, como `c:\temp\output.dat`?

★★★ **Exercício R11.4.** O que é uma linha de comando? Como um programa pode ler seus argumentos da linha de comando?

★★ **Exercício R11.5.** Forneça dois exemplos de programas em seu computador que leiam argumentos da linha de comando.

★★ **Exercício R11.6.** Se um programa Woozle for iniciado com o comando

```
java Woozle -Dname=piglet -I\eeyore -v heff.txt a.txt lump.txt
```

quais são os valores de `args[0]`, `args[1]` e assim por diante?

★★ **Exercício R11.7.** Qual é a diferença entre lançar uma exceção e capturar uma exceção?

★★ **Exercício R11.8.** O que é uma exceção verificada? O que é uma exceção não-verificada? Uma `NullPointerException` é uma exceção verificada ou não-verificada? Que exceções você precisa declarar com a palavra-chave `throws`?

★ **Exercício R11.9.** Por que você não precisa declarar que seu método talvez lance uma `NullPointerException`?

★★ **Exercício R11.10.** Quando seu programa executa um comando `throw`, qual instrução é executada em seguida?

★ **Exercício R11.11.** O que acontece se uma exceção não tiver uma cláusula `catch` correspondente?

★ **Exercício R11.12.** O que o seu programa pode fazer com o objeto de exceção que uma cláusula `catch` recebe?

★ **Exercício R11.13.** O tipo do objeto de exceção é sempre o mesmo tipo que o declarado na cláusula `catch` que o captura?

★ **Exercício R11.14.** Que tipo de valor você pode lançar? Você pode lançar uma string? Um inteiro?

★★ **Exercício R11.15.** Qual é o propósito da cláusula `finally`? Dê um exemplo de como ela pode ser utilizada.

★★★ **Exercício R11.16.** O que acontece quando uma exceção é lançada, o código de uma cláusula `finally` é executado e esse código lança uma exceção de um tipo diferente do original? Qual delas é capturada por uma cláusula `catch`? Escreva um programa de exemplo para experimentar isso.

★★ **Exercício R11.17.** Que exceções os métodos `next` e `nextInt` da classe `Scanner` podem lançar? Elas são exceções verificadas ou não-verificadas?

★★★ **Exercício R11.18.** Suponha que a cláusula `catch` no exemplo da Dica de Qualidade 11.3 tivesse sido deslocada para o bloco `try` interno, eliminando o bloco `try` externo. O código modificado funcionaria corretamente se (a) o construtor `FileReader` lançasse uma exceção e (b) o método `close` lançasse uma exceção?

★★ **Exercício R11.19.** Suponha que o programa na Seção 11.7 lesse um arquivo contendo os seguintes valores:

```
0
1
2
3
```

Qual é o resultado? Como o programa poderia ser aprimorado para fornecer um relatório de erro mais exato?

★★ **Exercício R11.20.** O método `readFile` na Seção 11.7 pode lançar uma `NullPointerException`? Se sim, como?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★★ **Exercício P11.1.** Escreva um programa que solicita ao usuário um nome de arquivo e imprime o número de caracteres, palavras e linhas nesse arquivo.

★★ **Exercício P11.2.** Escreva um programa que solicita ao usuário um nome de arquivo e conta o número de caracteres, palavras e linhas nesse arquivo. Então o programa pergunta o nome do próximo arquivo. Quando o usuário inserir um arquivo que não existe, o programa imprime as contagens totais de caracteres, palavras e linhas em todos os arquivos processados e sai.

★★ **Exercício P11.3.** Escreva um programa `CopyFile` que copia um arquivo para outro. Os nomes de arquivo são especificados na linha de comando. Por exemplo,

```
java CopyFile report.txt report.sav
```

★★ **Exercício P11.4.** Escreva um programa que *concatena* o conteúdo de vários arquivos em um arquivo único. Por exemplo,

```
java CatFiles chapter1.txt chapter2.txt chapter3.txt book.txt
```

cria um arquivo longo, `book.txt`, que contém o conteúdo dos arquivos `chapter1.txt`, `chapter2.txt` e `chapter3.txt`. O arquivo de saída é sempre o último arquivo especificado na linha de comando.

★★ **Exercício P11.5.** Escreva um programa `Find` que pesquisa todos os arquivos especificados na linha de comando e imprime todas as linhas contendo uma palavra-chave. Por exemplo, se você chamar

```
java Find ring report.txt address.txt Homework.java
```

então o programa pode imprimir

```
report.txt: has broken up an international ring of DVD bootleggers that
address.txt: Kris Kringle, North Pole
address.txt: Homer Simpson, Springfield
Homework.java: String filename;
```

A palavra-chave é sempre o primeiro argumento da linha de comando.

★★ **Exercício P11.6.** Escreva um programa que verifica a ortografia de todas as palavras em um arquivo. Você deve ler todas as palavras de um arquivo e verificar se estão contidas em uma lista de palavras. Uma lista de palavras está disponível na maioria dos sistemas UNIX no arquivo `/usr/dict/words`. (Se você não tiver acesso a um sistema UNIX, seu instrutor pode providenciar uma cópia para você.) O programa deve imprimir todas as palavras que ele não puder localizar na lista de palavras.

★★ **Exercício P11.7.** Escreva um programa que substitui cada linha de um arquivo pela mesma linha escrita de trás para frente. Por exemplo, se executar:

```
java Reverse HelloPrinter.java
```

então o conteúdo de `HelloPrinter.java` é modificado para:

```
retnirPolleH ssalc cilbup
{
)sgra ][gnirtS(niam diov citats cilbup
{
wodniw elosnoc eht ni gniteerg a yalpsiD //
;)"!dlroW ,olleH"(nltnirp.tuo.metsyS
}
}
```

Naturalmente, se você executar `Reverse` duas vezes no mesmo arquivo, você obtém o arquivo original de volta.

★★★ **Exercício P11.8.** Escreva um programa que substitui todos os caracteres de tabulação `'\t'` em um arquivo pelo número de espaços *apropriados*. Por padrão, a distância entre colunas de tabulação deve ser 3 (o valor que utilizamos neste livro para programas Java), mas ela pode ser alterada pelo usuário. Troque as tabulações pelo número de espaços necessários a fim de mudar para a próxima coluna de tabulação. Isso pode ser *menos* que três espaços. Por exemplo, considere a linha contendo `"\t|\t||\t|"`. A primeira tabulação é alterada para três espaços, a segunda para dois espaços e a terceira para um espaço. Seu programa deve ser executado como

```
java TabExpander nomeDoArquivo
```

ou

```
java -t larguraDaTabulação nomeDoArquivo
```

★ **Exercício P11.9.** Modifique a classe `BankAccount` para lançar uma `IllegalArgumentException` quando a conta for construída com um saldo negativo, quando um valor negativo for depositado ou quando um valor que não está entre 0 e o saldo atual é sacado. Escreva um programa de teste que faça com que as três exceções ocorram e que capture todas elas.

★★ **Exercício P11.10.** Repita o Exercício P11.9, mas lance exceções dos três tipos que você definir.

★★ **Exercício P11.11.** Escreva um programa que pede para o usuário inserir um conjunto de valores de ponto flutuante. Quando o usuário inserir um valor que não é um número, dê uma segunda chance para ele inserir o valor. Depois de duas chances, pare de ler a entrada. Adicione todos os valores corretamente especificados e imprima a soma quando o usuário terminar de inserir dados. Utilize o tratamento de exceções para detectar entradas impróprias.

★★ **Exercício P11.12.** Repita o Exercício P11.11, mas dê ao usuário quantas chances forem necessárias para inserir um valor correto. Feche o programa somente quando o usuário inserir uma entrada em branco.

★ **Exercício P11.13.** Modifique a classe `DataSetReader` para não chamar `hasNextInt` ou `hasNextDouble`. Simplesmente faça `nextInt` e `nextDouble` lançar uma `InputMismatchException` ou `NoSuchElementException` e capturá-la no método `main`.

★★ **Exercício P11.14.** Escreva um programa que lê um conjunto de descrições de moedas a partir de um arquivo. O arquivo de entrada tem o formato:

```
coinName1 coinValue1
coinName2 coinValue2
. . .
```

Adicione um método

```
void read(Scanner in) throws IOException
```

à classe `Coin`. Lance uma exceção se a linha atual não estiver adequadamente formatada. Então implemente um método

```
static ArrayList<Coin> readFile(String filename) throws IOException
```

No método `main`, chame `readFile`. Se uma exceção for lançada, dê ao usuário uma chance para selecionar outro arquivo. Se você ler todas as moedas com sucesso, imprima o valor total.

★★★ **Exercício P11.15.** Projete uma classe `Bank` que contenha uma quantidade de contas bancárias. Cada conta tem um número de conta e um saldo atual. Adicione um campo `accountNumber` à classe `BankAccount`. Armazene as contas bancárias em uma lista de arrays. Escreva um método `readFile` da classe `Bank` para ler um arquivo com o formato

```
accountNumber1  balance1
accountNumber2  balance2
. . .
```

Implemente os métodos `read` para as classes `Bank` e `BankAccount`. Escreva um programa de exemplo para ler um arquivo com contas bancárias, então imprima a conta com o saldo mais alto. Se o arquivo não estiver adequadamente formatado, dê ao usuário uma chance para selecionar outro arquivo.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 11.1.** Você pode ler o conteúdo de uma página Web com essa seqüência de comandos.

```
String address = "http://java.sun.com/index.html";
URL u = new URL(address);
URLConnection connection = u.openConnection();
InputStream stream = connection.getInputStream();
Scanner in = new Scanner(stream);
. . .
```

Alguns desses métodos podem lançar exceções – consulte a documentação da API. Projete uma classe `LinkFinder` que localiza todos os hyperlinks na forma

```
<a href="link">texto do link</a>
```

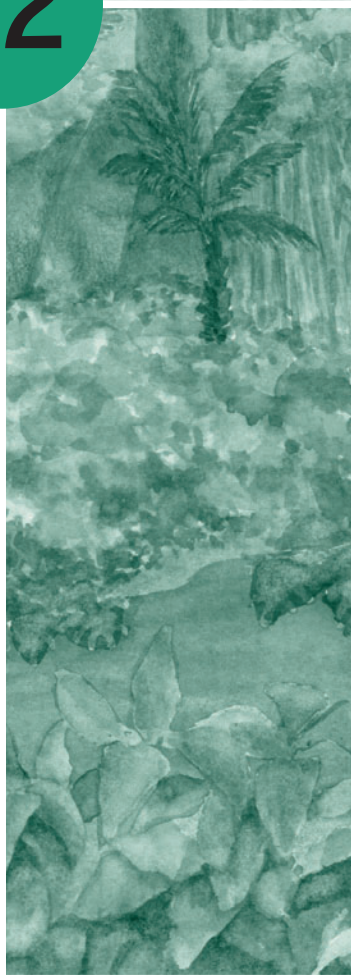
Lance uma exceção se localizar um hyperlink malformado. Você ganha um crédito extra se seu programa puder seguir os links que ele descobre e se localizar os links nessas páginas da Web também. (Esse é o método que sistemas de pesquisa como o Google utilizam para localizar sites Web.)

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Quando o objeto `PrintWriter` é criado, o arquivo de saída é esvaziado. Infelizmente, esse é o próprio arquivo de entrada. O arquivo de entrada agora está vazio e o laço `while` termina imediatamente.
2. O programa lança e captura uma `FileNotFoundException`, imprime uma mensagem de erro e termina.
3. Lance uma exceção se o valor sendo depositado for menor que zero.
4. O saldo ainda é zero porque a última instrução do método `withdraw` nunca foi executada.
5. A especificação `throws IOException` é suficiente porque `FileNotFoundException` é uma subclasse de `IOException`.
6. Porque os programadores devem simplesmente verificar os ponteiros `null` em vez de tentar tratar uma `NullPointerException`.
7. O construtor `FileReader` é bem-sucedido e `in` é construído. Então a chamada `in.next()` lança uma `NoSuchElementException` e o bloco `try` é abortado. Nenhuma das cláusulas catch corresponde, portanto, nenhuma é executada. Se nenhuma das chamadas de método no código capturar a exceção, o programa termina.
8. Não – você captura ambas da mesma maneira, como mostra o exemplo de código na página 462. Lembre-se de que `IOException` é uma exceção verificada e `NumberFormatException` é não-verificada.
9. Se ela fosse declarada dentro do bloco `try`, seu escopo só se estenderia até o fim do bloco `try` e a cláusula `finally` não poderia fechá-lo.
10. O construtor `FileReader` lança uma exceção. A cláusula `finally` é executada. Como `reader` é `null`, a chamada a `close` não é executada. Em seguida, uma cláusula `catch` que corresponde à `FileNotFoundException` é localizada. Se nenhuma existir, o programa termina.
11. Passar a string de mensagem de exceção para a superclasse `RuntimeException`.
12. Tanto `Exception` como `IOException` são boas escolhas. Como a corrupção de arquivo está além do controle do programador, essa deve ser uma exceção verificada, então seria errado estender `RuntimeException`.
13. Ele não poderia fazer muito com elas. A classe `DataSetReader` é uma classe reutilizável que pode ser usada para sistemas com diferentes linguagens e diferentes interfaces com o usuário. Portanto, ela não pode estabelecer um diálogo com o usuário do programa.
14. `DataAnalyzer.main` chama `DataSetReader.readFile`, que chama `readData`. A chamada `in.hasNextInt()` retorna `false` e `readData` lança uma `BadDataException`. O método `readFile` não a captura, então ela é propagada de volta para `main`, onde é capturada.

Capítulo 12

Projeto Orientado a Objetos



Objetivos do capítulo

- Aprender sobre o ciclo de vida de um software
- Aprender a descobrir novas classes e métodos
- Entender o uso de cartões CRC para descoberta de classe
- Ser capaz de identificar herança, agregação e relacionamentos de dependência entre as classes
- Dominar o uso de diagramas de classe UML para descrever relacionamentos de classe
- Aprender a utilizar projeto orientado a objetos para construir programas complexos

Para implementar um sistema de software com sucesso, seja ele tão simples quanto o seu próximo dever de casa ou tão complexo quanto o sistema de monitoramento de tráfego aéreo, é necessário investimento em planejamento, projeto e teste. Na verdade, em projetos maiores, a quantidade de tempo gasta em planejamento é muito maior que a quantidade de tempo gasta em programação e testes.

Se você descobrir que a maior parte do tempo dispendido com suas tarefas de casa é na frente do computador, digitando código e corrigindo erros, provavelmente está gastando mais tempo nisso do que deveria. Você poderia reduzir esse tempo total investindo mais na fase de planejamento e projeto. Este capítulo mostra como abordar essas tarefas de maneira sistemática, utilizando a metodologia de projeto orientada a objetos.

Conteúdo do Capítulo

12.1 O ciclo de vida de um software 480
Fato aleatório 12.1: Produtividade do programador ⊕
12.2 Descobrindo classes 485
12.3 Relacionamentos entre classes 488
Como fazer 12.1: Cartões CRC e diagramas UML 490
Tópico avançado 12.1: Atributos e métodos em diagramas UML ⊕
Tópico avançado 12.2: Multiplicidades ⊕
Tópico avançado 12.3: Agregação e associação ⊕
12.4 Estudo de caso: Imprimindo uma fatura 491
12.5 Estudo de caso: Um caixa automático 503
Fato aleatório 12.2: Desenvolvimento de software – arte ou ciência? ⊕

12.1 O ciclo de vida de um software

O ciclo de vida de um software engloba todas as atividades, da análise inicial até a obsolescência.

Um processo formal de desenvolvimento de software descreve as fases do processo de desenvolvimento e fornece diretrizes sobre como executá-las.

Nesta seção, discutiremos o ciclo de vida do software: as atividades que acontecem desde o momento em que um software é concebido até o momento em que ele é, por fim, aposentado.

Normalmente, um projeto de software é iniciado porque um cliente tem um problema e está disposto a pagar para resolvê-lo. O Departamento de Defesa Norte-Americano, o cliente de muitos projetos de programação, foi um dos primeiros proponentes de um *processo formal* para desenvolvimento de software. Um processo formal identifica e descreve diferentes fases, dá diretrizes para executá-las e indica quando avançar de uma fase para a próxima.

Muitos engenheiros de software dividem o processo de desenvolvimento nas cinco fases a seguir:

- Análise
- Projeto
- Implementação
- Teste
- Implantação

Na fase de *análise*, você decide *o que* o projeto deve fazer; não pensa sobre *como* o programa realizará suas tarefas. O resultado da fase de análise é um *documento de requisitos*, que descreve detalhadamente o que o programa será capaz de fazer depois de concluído. Parte desse documento de requisitos pode ser um manual de usuário que informa como o usuário operará o programa para obter os benefícios prometidos. Outra parte apresenta os critérios de desempenho – quantas entradas o programa deve tratar em determinado tempo ou quais são os seus requisitos de armazenamento e memória máximos.

Na fase de *projeto*, ou *design*, você desenvolve um plano de como implementar o sistema. Você descobre as estruturas que fundamentam o problema a ser resolvido. Ao utilizar um projeto orientado a objetos, você decide quais classes são necessárias e quais

são seus métodos mais importantes. O resultado dessa fase é uma descrição das classes e métodos, com diagramas que mostram os relacionamentos entre as classes.

Na fase de *implementação*, você escreve e compila o código do programa para implementar as classes e métodos que foram descobertos na fase de projeto. O resultado dessa fase é o programa completo.

Na fase de *teste*, você executa os testes para verificar se o programa funciona corretamente. O resultado dessa fase é um relatório que descreve os testes que você fez e seus resultados.

Na fase de *implantação*, os usuários instalam e utilizam o programa de acordo com sua finalidade.

O modelo de desenvolvimento de software em cascata descreve um processo seqüencial de análise, projeto, implementação, teste e implantação.

Quando processos de desenvolvimento formal foram estabelecidos pela primeira vez, no início da década de 1970, os engenheiros de software tinham um modelo visual muito simples dessas fases. Eles postulavam que uma fase executaria até a sua conclusão, e então sua saída *transbordaria* para a próxima fase e esta iniciaria. Esse modelo é chamado de modelo de desenvolvimento de software *em cascata* (veja Figura 1).

Em um mundo ideal, o modelo em cascata tem um grande apelo: você imagina o que fazer; depois como fazer; então faz; em seguida, verifica se o fez corretamente; e, por fim, passa o produto para o cliente. Mas quando era aplicado rigidamente, o modelo em cascata simplesmente não funcionava. Era muito difícil elaborar uma especificação de requisitos perfeita. Era muito comum descobrir na fase de projeto que os requisitos eram inconsistentes ou que uma pequena alteração nos requisitos levaria a um sistema que era mais fácil projetar e mais útil para o cliente, mas a fase de análise já havia passado, então o projetista não tinha escolha – tinha que adotar os requisitos existentes, incluindo erros e tudo mais. Esse problema iria se repetir durante

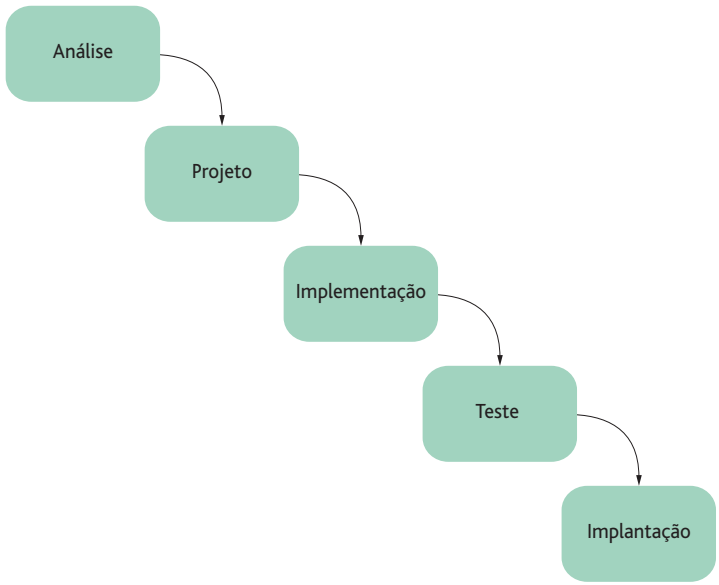


Figura 1 Modelo em cascata.

a implementação. Talvez os projetistas pensassem que sabiam resolver o problema com a maior eficiência possível, mas quando o projeto era realmente implementado, descobriam que o programa resultante não era tão rápido quanto eles haviam pensado. Certamente você está familiarizado com a próxima transição. Quando o programa era passado para o departamento de controle de qualidade para os testes, muitos erros localizados seriam melhor corrigidos reimplementando o programa, ou mesmo reprojetando-o, mas o modelo em cascata não permitia isso. Por fim, quando os clientes recebiam o produto final, muitas vezes ficavam insatisfeitos. Embora fosse comum um grande envolvimento dos clientes na fase de análise, em geral eles não sabiam exatamente do que precisavam. Afinal de contas, pode ser muito difícil descrever como você quer utilizar um produto que nunca viu antes. Mas quando os clientes começavam a utilizar o programa, acabavam percebendo o que seria melhor. É claro que era tarde demais e então tinham de conviver com o que receberam.

O modelo espiral de desenvolvimento de software descreve um processo iterativo em que projeto e implementação são repetidos.

Claramente é necessário ter algum nível de iteração. Simplesmente deve haver um mecanismo para lidar com erros da fase precedente. Um *modelo espiral,* originalmente proposto por Barry Boehm em 1988, divide o processo de desenvolvimento em várias fases (veja Figura 2). As fases iniciais focam a construção de *protótipos.* Protótipo é um pequeno sistema que mostra alguns aspectos do sistema final. Como os protótipos modelam somente uma parte de um sistema e não precisam suportar abusos por parte do cliente, eles podem ser implementados rapidamente. É comum construir um *protótipo de interface com o usuário* que mostra a interface com o usuário em ação. Isso dá uma chance inicial de os clientes familiarizarem-se com o sistema e sugerirem melhoras antes da conclusão da análise. Outros protótipos podem ser construídos para validar interfaces com sistemas externos, para testar o desempenho e assim por diante. As lições aprendidas a partir do desenvolvimento de um protótipo podem ser aplicadas à próxima iteração da espiral.

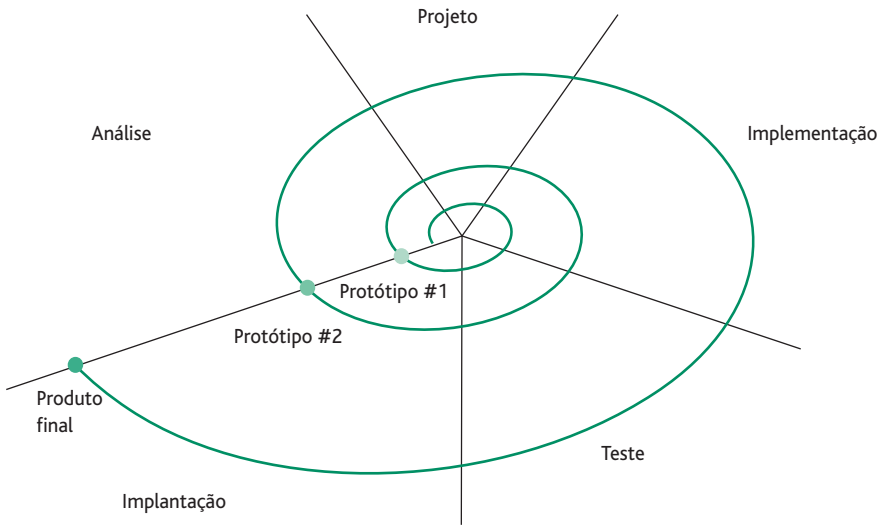


Figura 2 Modelo espiral.

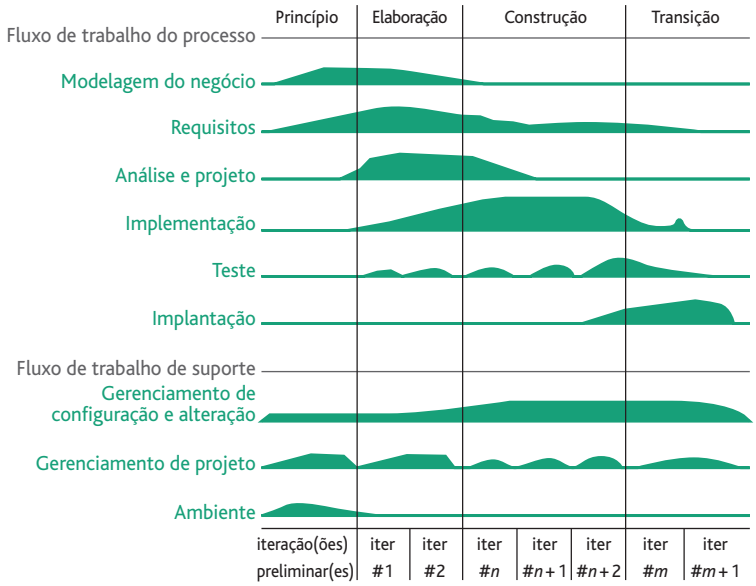


Figura 3 Níveis de atividade na metodologia *Rational Unified Process* [1].

Baseando-se em repetidas avaliações e realimentações, um processo de desenvolvimento que segue o modelo espiral tem uma chance maior de apresentar um sistema satisfatório. Entretanto, também existe um perigo. Se os engenheiros acreditam que não precisam fazer um bom trabalho, pois sempre podem fazer outra iteração, então haverá muitas iterações e a conclusão do processo exigirá muito mais tempo.

A Programação Extrema (eXtreme Programming – XP) é uma metodologia de desenvolvimento que se esforça para obter simplicidade eliminando a estrutura formal e se concentrando nas melhores práticas.

A Figura 3 mostra níveis de atividade no "Processo Unificado Racional" (Rational Unified Process), uma metodologia de processo de desenvolvimento proposta pelos criadores da UML. Os detalhes não são importantes, mas como você pode ver, esse é um processo complexo, envolvendo múltiplas iterações.

Mesmo os processos de desenvolvimento complexos com muitas iterações nem sempre obtêm sucesso. Em 1999, Kent Beck publicou um livro influente [2] sobre *Programação Extrema*, uma metodologia de desenvolvimento que se esforça para obter simplicidade eliminando a maioria dos acessórios formais de uma metodologia de desenvolvimento tradicional e se concentrando em um conjunto de *práticas:*

- *Planejamento realista:* os clientes devem tomar decisões de negócio, os programadores devem tomar decisões técnicas. Atualize o plano quando ele entrar em conflito com a realidade.
- *Versões pequenas:* lance um sistema útil rapidamente, então lance atualizações em um ciclo muito curto.
- *Metáfora:* todos os programadores devem compartilhar uma história simples que explique o sistema sob desenvolvimento.
- *Simplicidade:* projete para que tudo seja o mais simples possível em vez de preparar para uma futura complexidade.

- *Teste:* tanto programadores como clientes devem escrever casos de teste. O sistema é continuamente testado.
- *Refatoração:* os programadores devem reestruturar o sistema continuamente para aprimorar o código e eliminar duplicação.
- *Programação em duplas:* faça duplas de programadores e exija que cada um escreva código em um único computador.
- *Propriedade coletiva:* todos os programadores têm permissão de alterar inteiramente o código conforme a necessidade.
- *Integração contínua:* sempre que uma tarefa for completada, monte o sistema inteiro e teste.
- *Semana de 40 horas:* não esconda agendas irreais com rajadas de esforços heróicos.
- *Cliente no local:* um cliente real do sistema deve estar acessível aos membros da equipe o tempo todo.
- *Padrões de codificação:* os programadores devem seguir padrões que enfatizam a autodocumentação do código.

Muitas dessas práticas são de bom senso. Outras, como o requisito de programação em dupla, são surpreendentes. Beck alega que o valor da abordagem *Extreme Programming* reside na sinergia dessas práticas – a soma é maior que as partes.

Em seu primeiro curso de programação, você não desenvolverá sistemas tão complexos que precisem de uma metodologia completa para resolver os problemas dos seus deveres de casa. Essa introdução ao processo de desenvolvimento deve, porém, mostrar que o desenvolvimento de software bem-sucedido envolve mais do que apenas simples codificação. No restante deste capítulo, faremos um exame mais minucioso da *fase de projeto* do processo de desenvolvimento de software.

Autoverificação da aprendizagem

1. Suponha que você assine um contrato, prometendo que irá, por um preço combinado, projetar, implementar e testar um pacote de software exatamente como especificado em um documento de requisitos. Qual é o principal risco que você e seu cliente enfrentarão nesse cenário?
2. A programação extrema segue um modelo em cascata ou um modelo espiral?
3. Qual é o propósito do "cliente no local" da programação extrema?

Fato aleatório 12.1

Produtividade do programador

O Fato Aleatório 12.1 apresenta informações sobre a produtividade de programadores individuais e equipes.

12.2 Descobrindo classes

Na fase de projeto do desenvolvimento de software, sua tarefa é descobrir estruturas que possibilitem a implementação de um conjunto de tarefas em um computador. Ao utilizar o processo de projeto orientado a objetos, você tem as seguintes tarefas:

No projeto orientado a objetos, você descobre classes, determina as responsabilidades das classes e descreve os relacionamentos entre elas.

1. Descobrir as classes.
2. Determinar as responsabilidades de cada classe.
3. Descrever os relacionamentos entre as classes.

Uma classe representa um conceito útil. Você viu classes para entidades concretas, como contas bancárias, elipses e produtos. Outras classes representam conceitos abstratos, como fluxos e janelas.

Uma regra simples para localizar classes é procurar *substantivos* na descrição da tarefa. Por exemplo, suponha que seu trabalho seja imprimir uma fatura (invoice) como a mostrada na Figura 4. As classes óbvias que vêm à mente são `Invoice` (fatura), `LineItem` (item) e `Customer` (cliente). É uma boa idéia manter uma lista de *classes candidatas* em um quadro branco ou uma folha de papel. Enquanto pensa em uma solução, simplesmente coloque todas as idéias sobre classes na lista. Depois você sempre pode eliminar as que não foram úteis.

INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	$29.95	$89.85
Hair Dryer	1	$24.95	$24.95
Car Vacuum	2	$19.99	$39.98

AMOUNT DUE: **$154.78**

Figura 4 Uma fatura.

Ao localizar classes, tenha os seguintes pontos em mente:

- Uma classe representa um conjunto de objetos com o mesmo comportamento. Entidades com múltiplas ocorrências na sua descrição de problema, como clientes ou produtos, são bons candidatos a objetos. Descubra o que elas têm em comum e projete classes para capturar esses pontos em comum.
- Algumas entidades devem ser representadas como objetos, outras como tipos primitivos. Por exemplo, um endereço deve ser um objeto de uma classe `Address` ou deve ser simplesmente uma string? Não há resposta perfeita – depende da tarefa que você quer resolver. Se seu software precisa analisar endereços (por exemplo, para determinar custos de entrega), então uma classe `Address` é um projeto apropriado. Mas se o software nunca precisar dessa capacidade, você não deve desperdiçar tempo em um projeto complexo demais. É seu trabalho descobrir um projeto equilibrado, que não seja limitado nem excessivamente geral.
- Nem todas as classes podem ser descobertas na fase de análise. A maioria dos programas complexos precisa de classes para propósitos táticos, como acesso a arquivo ou banco de dados, interfaces com o usuário, mecanismos de controle e assim por diante.
- Algumas das classes necessárias já podem existir na biblioteca padrão ou em um programa que você desenvolveu anteriormente. Você também pode ser capaz de usar herança para estender classes existentes para classes que atendam suas necessidades.

Assim que um conjunto de classes é identificado, você precisa definir o comportamento de cada uma. Isto é, você precisa descobrir quais métodos cada objeto precisa aplicar para resolver o problema de programação. Uma regra simples para localizar esses métodos é procurar por *verbos* que descrevam a tarefa e então relacionar os verbos aos objetos apropriados. Por exemplo, no programa de fatura, uma classe precisa calcular o valor devido. Agora você precisa descobrir *qual classe* é responsável por esse método. Os clientes calculam o que eles devem? As faturas somam o valor devido? Os itens somam a si próprios? A melhor escolha é tornar o "cálculo do valor devido" uma responsabilidade da classe `Invoice`.

Um cartão CRC descreve uma classe, suas responsabilidades e suas classes colaboradoras.

Uma maneira excelente de executar essa tarefa é o "método de ficha CRC". *CRC* significa "*c*lasses", "*r*esponsabilidades", "*c*olaboradores" e, em sua forma mais simples, o método funciona da seguinte maneira. Utilize uma ficha para cada *classe* (Veja Figura 5). Enquanto pensa nos verbos de descrição de tarefa que indicam os métodos, você pega a ficha da classe que você considera responsável e escreve essa *responsabilidade* na ficha.

Para cada responsabilidade, você registra as outras classes necessárias para cumpri-la. Essas classes são os colaboradores.

Por exemplo, suponha que você decida que uma fatura deve calcular o valor devido. Então escreva "calcular o valor devido" no lado esquerdo de uma ficha com o título `Invoice`.

Se uma classe pode executar essa responsabilidade sozinha, não faça mais nada. Mas se a classe precisar da ajuda de outras classes, anote os nomes desses colaboradores no lado direito da ficha.

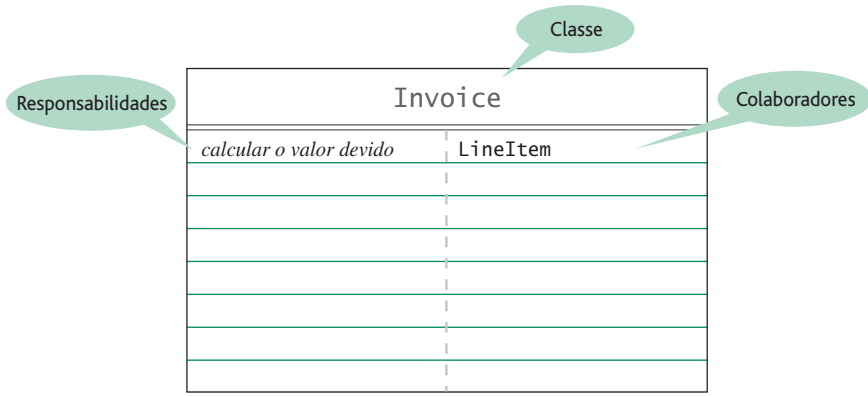


Figura 5 Uma ficha CRC.

Para calcular o total, a fatura precisa perguntar a cada item de linha seu preço total. Portanto, a classe `LineItem` é um colaborador.

Essa é uma boa hora para consultar a ficha da classe `LineItem`. Ela tem um método "obter preço total"? Se não tiver, adicione um.

Como saber que você está no caminho certo? Para cada responsabilidade, pergunte a si mesmo como isso pode realmente ser feito, utilizando as responsabilidades escritas nas várias fichas. Muitas pessoas acham útil agrupar as fichas em uma tabela, para que os colaboradores fiquem próximos uns dos outros, e simular tarefas movendo um indicador (como uma moeda) de uma ficha para a próxima para indicar qual objeto está atualmente ativo.

Não se esqueça que as responsabilidades que você lista na ficha CRC estão em um *nível alto*. Às vezes uma única responsabilidade pode precisar de dois ou mais métodos Java para executá-la. Alguns pesquisadores dizem que uma ficha CRC não tem mais do que três responsabilidades distintas.

O método da ficha CRC é propositadamente informal, para que você possa ser criativo e para que possa descobrir classes e suas propriedades. Assim que achar que estabeleceu um bom conjunto de classes, você precisará saber como elas se relacionam entre si. É possível localizar classes com propriedades comuns, para que algumas responsabilidades possam ser tratadas por uma superclasse comum? É possível organizar classes em grupos que sejam independentes um do outro? Descobrir relacionamentos entre classes e documentá-los com diagramas é o tópico da próxima seção.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

4. Suponha que a fatura deva ser salva em um arquivo. Nomeie um possível colaborador.
5. Examinando a fatura na Figura 4, qual é uma possível responsabilidade da classe `Customer`?
6. O que você faz se uma ficha CRC tiver dez responsabilidades?

12.3 Relacionamentos entre classes

Ao projetar um programa, é útil documentar os relacionamentos entre classes. Isso ajuda de várias maneiras. Por exemplo, se descobrir classes com comportamentos comuns, você pode poupar esforços colocando-os em uma superclasse. Se souber que algumas classes *não* têm relação entre si, você pode designar diferentes programadores para implementar cada uma delas, sem se preocupar com que um deles tenha de esperar pelo outro.

Você já viu o relacionamento de herança entre classes muitas vezes neste livro. A herança é um relacionamento muito importante, mas, como constatamos, não é o único relacionamento útil e pode estar sendo usada em excesso.

A herança é um relacionamento entre uma classe mais geral (a superclasse) e uma classe mais especializada (a subclasse). Esse relacionamento é, muitas vezes, descrito como o relacionamento *é-um*. Todo caminhão é um veículo. Toda conta poupança é uma conta bancária. Todo círculo é uma elipse (com largura e altura iguais).

A herança (o relacionamento *é-um*) às vezes é utilizada inadequadamente quando o relacionamento *tem-um* seria mais apropriado.

Entretanto, às vezes há um uso abusivo da herança. Por exemplo, considere uma classe `Tire` que descreve um pneu de carro. A classe `Tire` deve ser uma subclasse de uma classe `Circle`? Parece conveniente. Há muitos métodos úteis na classe `Circle` – por exemplo, a classe `Tire` pode herdar métodos que calculam o raio, o perímetro e o ponto central, o que deve ser útil ao desenhar formas de pneu. Embora possa ser conveniente para o programador, essa combinação não faz sentido conceitualmente. Não é verdade que cada pneu é um círculo. Pneus são partes do carro, enquanto círculos são objetos geométricos. Entretanto, existe um relacionamento entre pneus e círculos. Um pneu *tem um* círculo como seu limite. Java permite modelar esse relacionamento também. Utilize um campo de instância:

```
public class Tire
{
   . . .
   private String rating;
   private Circle boundary;
}
```

O termo técnico para esse relacionamento é a *agregação*. Cada `Tire` agrega um objeto `Circle`. Em geral, uma classe agrega outra classe se seus objetos tiverem objetos da outra classe.

A agregação (o relacionamento *tem-um*) indica que objetos de uma classe contêm referências a objetos de outra classe.

Eis outro exemplo. Cada carro *é um* veículo. Cada carro *tem um* pneu (de fato, tem quatro ou, se contarmos o estepe, cinco). Portanto, você utilizaria a herança a partir de `Vehicle` e utilizaria a agregação de objetos `Tire`:

```
public class Car extends Vehicle
{
   . . .
   private Tire[] tires;
}
```

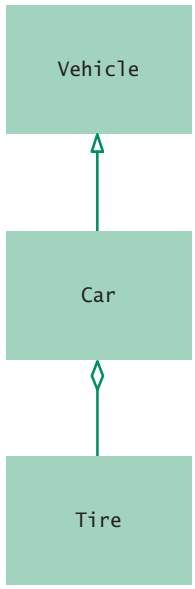


Figura 6
Notação UML para herança e agregação.

Neste livro, utilizamos a notação UML para representar diagramas de classe. Você já viu muitos exemplos da notação UML para herança – uma seta com um triângulo aberto apontando para a superclasse. Na notação UML, a agregação é representada por uma linha sólida com um símbolo na forma de losango ao lado da classe agregadora. A Figura 6 mostra um diagrama de classes com um relacionamento de herança e um de agregação.

Dependência é outro nome para o relacionamento de uso.

O relacionamento de agregação está ligado ao relacionamento de dependência, discutido no Capítulo 8. Lembre-se de que uma classe depende da outra se um de seus métodos *usa* um objeto da outra classe de alguma maneira.

Por exemplo, muitos de nossos aplicativos dependem da classe `Scanner`, porque utilizam um objeto `Scanner` para ler entrada.

A agregação é uma forma mais forte de dependência. Se uma classe tiver objetos de outra classe, ela certamente usa a outra classe. Entretanto, o inverso não é verdadeiro. Por exemplo, uma classe pode usar a classe `Scanner` sem nunca definir um campo de instância da classe `Scanner`. A classe pode simplesmente construir uma variável local do tipo `Scanner` ou seus métodos podem receber objetos `Scanner` como parâmetros. Esse uso não é agregação porque os objetos da classe não contêm objetos `Scanner` – eles apenas os criam ou os recebem pela duração de um método individual.

Você precisa ser capaz de distinguir as notações UML para herança, implementação de interface, agregação e dependência.

Geralmente, a agregação é necessária quando um objeto precisa lembrar de outro *entre chamadas de método*.

Como visto no Capítulo 8, a notação UML para dependência é uma linha tracejada com uma seta aberta que aponta para a classe dependente.

As setas na notação UML podem nos confundir. A Tabela 1 mostra um resumo dos quatro símbolos de relacionamento UML que utilizamos neste livro.

Tabela 1 Símbolos de relacionamento UML

Relacionamento	Símbolo	Estilo da linha	Tipo da ponta da seta
Herança	——▷	Sólido	Triângulo
Implementação de Interface	- - - - ▷	Pontilhado	Triângulo
Agregação	◇——	Sólido	Losango
Dependência	- - - - >	Pontilhado	Aberto

Autoverificação da aprendizagem

7. Considere as classes `Bank` e `BankAccount` do Capítulo 7. Como elas se relacionam?
8. Considere os objetos `BankAccount` e `SavingsAccount` do Capítulo 10. Como eles se relacionam?
9. Considere a classe `BankAccountTester` do Capítulo 3. De que classes ela depende?

Como fazer 12.1

Cartões CRC e diagramas UML

Antes de escrever código para um problema complexo, você precisa projetar uma solução. A metodologia introduzida neste capítulo sugere que você siga um processo de projeto composto das seguintes tarefas:

1. Descubra as classes.
2. Determine as responsabilidades de cada classe.
3. Descreva os relacionamentos entre as classes.

As fichas CRC e os diagramas UML ajudam a descobrir e registrar essas informações.

Passo 1 Descubra as classes.

Destaque os substantivos na descrição de problema. Faça uma lista de substantivos. Risque aqueles que não parecem candidatos razoáveis a classes.

Passo 2 Descubra as responsabilidades.

Faça uma lista das principais tarefas que seu sistema precisa cumprir. Dessas tarefas, selecione uma que não seja trivial e que seja intuitiva para você. Localize uma classe responsável por executar essa tarefa. Faça uma ficha e escreva o nome e a tarefa dela. Agora pergunte a você mesmo como um objeto da classe pode executar a tarefa. Provavelmente ele precisa da ajuda de outros objetos. Então faça a ficha CRC das classes às quais esses objetos pertencem e escreva as responsabilidades deles.

Não tenha receio de eliminar, mover, dividir ou mesclar responsabilidades. Descarte as fichas que se tornarem muito confusas. Esse é um processo informal.

Seu trabalho termina quando você passar por todas as tarefas importantes e achar que elas podem ser inteiramente resolvidas com as classes e responsabilidades que você descobriu.

Passo 3 Descreva os relacionamentos.

Faça um diagrama de classes que mostre os relacionamentos entre todas as classes que você descobriu.

Inicie com herança – o relacionamento *é-um* entre classes. Toda classe é uma especialização de outra classe? Se for, desenhe setas de herança. Tenha em mente que muitos projetos, especialmente os de programas simples, não utilizam herança extensamente.

A coluna "colaboradores" das fichas CRC informa quais classes utilizam outras. Desenhe setas de uso para os colaboradores nas fichas CRC.

Alguns relacionamentos de dependência dão origem a agregações. Para cada um dos relacionamentos de dependência, pergunte a você mesmo: como o objeto localiza seu colaborador? Ele navega diretamente para ele porque armazena uma referência? Nesse caso, desenhe uma seta de agregação. Ou o colaborador é um parâmetro de um método ou um valor de retorno? Então simplesmente desenhe uma seta de dependência.

TÓPICOS AVANÇADOS 12.1 – 12.3

Notação UML

Os Tópicos Avançados 12.1–12.3 discutem recursos avançados da notação UML: atributos e métodos em diagramas UML, multiplicidades e agregação e associação.

12.4 Estudo de caso: Imprimindo uma fatura

Neste capítulo, discutimos um processo de desenvolvimento, composto de cinco partes, que é particularmente muito apropriado para programadores iniciantes:

1. Colete os requisitos.
2. Utilize fichas CRC para localizar classes, responsabilidades e colaboradores.
3. Utilize os diagramas UML para registrar relacionamentos de classe.
4. Utilize `javadoc` para documentar comportamentos de métodos.
5. Implemente seu programa.

Não há muitas notações para aprender. Os diagramas de classe são simples de desenhar. O produto final da fase de projeto é obviamente útil para a fase de implementação – você simplesmente pega os arquivos-fonte e começa a adicionar o código do método. É claro que, quando seus projetos tornarem-se mais complexos, você precisará aprender mais sobre métodos de projeto formais. Há muitas técnicas para descrever os cenários de objeto, o seqüenciamento de chamada, a estrutura de larga escala de programas e assim por diante, que são muito benéficos mesmo para projetos relativamente simples. O livro *Unified Modeling Language User Guide* [1] dá uma boa visão geral dessas técnicas.

Nesta seção, apresentaremos, passo a passo, a técnica de projeto orientado a objetos com um exemplo muito simples. Nesse caso, a metodologia pode parecer exagerada, mas é uma boa introdução à mecânica de cada passo. Assim, você estará melhor preparado para o exemplo mais complexo a seguir.

12.4.1 Requisitos

A tarefa desse programa é imprimir uma fatura. Uma fatura descreve os custos de um conjunto de produtos em certas quantidades. (Omitimos complexidades como datas, impostos, número da fatura e número do cliente.) O programa simplesmente imprime o endereço de cobrança, todos os itens de linha e o valor devido. Cada item de linha contém a descrição e o preço unitário de um produto, a quantidade encomendada e o preço total.

```
                              I N V O I C E

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description                Price   Qty   Total
Toaster                    29.95    3    89.85
Hair dryer                 24.95    1    24.95
Car vacuum                 19.99    2    39.98

AMOUNT DUE: $154.78
```

Além disso, por questão de simplicidade, não fornecemos uma interface com o usuário. Apenas fornecemos um programa de teste que adiciona os itens de linha à fatura e então a imprime.

12.4.2 Fichas CRC

Primeiro, você precisa descobrir as classes. As classes correspondem a substantivos na descrição de requisitos. Nesse problema, é muito óbvio quais são os substantivos:

```
Invoice (Fatura)
Address (Endereço)
LineItem (Item)
Product (Produto)
Description (Descrição)
Price (Preço)
Quantity (Quantidade)
Total (Total)
Amount Due (Valor devido)
```

(Naturalmente, `Toaster` não conta – é a descrição de um objeto `LineItem` e, portanto, um valor de dados, não o nome de uma classe.)

Descrição e preço são campos da classe `Product`. E a quantidade? A quantidade não é um atributo de um `Product`. Assim como na fatura impressa, vamos ter uma classe `LineItem` que registra o produto e a quantidade (como "3 torradeiras").

O total e o valor devido são calculados – não armazenados em qualquer lugar. Portanto, eles não são classes.

Depois desse processo de eliminação, restam quatro candidatos a classes:

```
Invoice
Address
LineItem
Product
```

Cada um deles representa um conceito útil, portanto, transformaremos todos em classes.

O propósito do programa é imprimir uma fatura. Entretanto, a classe `Invoice` não necessariamente saberá se exibe a saída em `System.out`, em uma área de texto ou em um arquivo. Portanto, vamos relaxar um pouco a tarefa e tornar a fatura responsável pela *formatação* da fatura. O resultado é uma string (que contém múltiplas linhas) que pode ser impressa ou exibida. Registre essa responsabilidade em uma ficha CRC:

Invoice
formatar a fatura

Como uma fatura formata a si própria? Ela deve formatar o endereço de cobrança, todos os itens de linha e então adicionar o valor devido. Como a fatura pode formatar um endereço? Não pode – essa responsabilidade é da classe `Address`. Isso nos leva a uma segunda ficha CRC:

Address
formatar o endereço

De maneira semelhante, a formatação de um item de linha é responsabilidade da classe `LineItem`.

O método `format` da classe `Invoice` chama os métodos `format` das classes `Address` e `LineItem`. Sempre que um método usa outra classe, você lista essa outra classe como um colaborador. Em outras palavras, `Address` e `LineItem` são colaboradores de `Invoice`:

Invoice	
formatar a fatura	`Address`
	`LineItem`

Durante a formatação da fatura, ela também precisa calcular o valor devido total. Para obter esse valor, ela deve perguntar o preço total do item a cada item de linha.

Como um item de linha obtém esse total? Você deve perguntar ao produto o preço unitário e então multiplicá-lo pela quantidade. Isto é, a classe `Product` deve revelar o preço unitário e ser colaboradora da classe `LineItem`.

Por fim, a fatura deve ser preenchida com produtos e quantidades, de modo que a formatação do resultado faça sentido. Essa também é uma responsabilidade da classe `Invoice`.

Temos agora um conjunto de fichas CRC que completam o processo de ficha CRC.

Product
obter descrição
obter preço unitário

LineItem	
formatar o item	`Product`
obter preço total	

Invoice	
formatar a fatura	Address
adicionar um produto e quantidade	LineItem
	Product

12.4.3 Diagramas UML

Os relacionamentos de dependência vêm da coluna de colaboração nas fichas CRC. Cada classe depende das classes com as quais ela colabora. Em nosso exemplo, a classe `Invoice` colabora com as classes `Address`, `LineItem` e `Product`. A classe `LineItem` colabora com a classe `Product`.

Agora pergunte a você mesmo quais dessas dependências são realmente agregações. Como uma fatura conhece os objetos endereço, item de linha e produto com os quais ela colabora? Um objeto fatura deve armazenar referências aos endereços e itens de linha ao formatar a fatura. Mas um objeto fatura não precisa armazenar uma referência a um objeto produto ao adicionar um produto. O produto transforma-se em um item de linha e então é responsabilidade do item armazenar uma referência a ele.

Portanto, a classe `Invoice` agrega as classes `Address` e `LineItem`. A classe `LineItem` agrega a classe `Product`. Entretanto, não existe o relacionamento *tem-um* entre uma fatura e um produto. Uma fatura não armazena produtos diretamente – eles são armazenados nos objetos `LineItem`.

Não há nenhuma herança nesse exemplo.

A Figura 7 mostra os relacionamentos de classe que descobrimos.

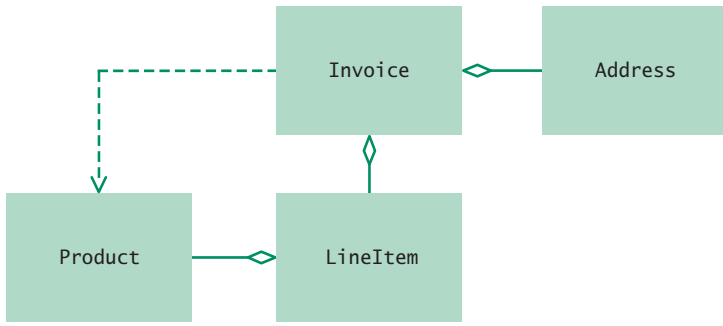


Figura 7 Relacionamentos entre as classes da fatura.

12.4.4 Documentação de métodos

Utilize comentários no estilo `javadoc` (com o corpo dos métodos deixados em branco) para registrar o comportamento das classes.

O passo final da fase de projeto é escrever a documentação das classes e métodos descobertos. Simplesmente escreva um arquivo-fonte Java para cada classe, escreva os comentários dos métodos que você descobriu e deixe o corpo em branco.

```
/**
   Descreve uma fatura para um conjunto de produtos comprados.
*/
public class Invoice
{
   /**
      Adiciona uma taxa sobre um produto a essa fatura.
      @param aProduct produto que o cliente encomendou
      @param quantity quantidade do produto
   */
   public void add(Product aProduct, int quantity)
   {
   }

   /**
      Formata a fatura.
      @return fatura formatada
   */
   public String format()
   {
   }
}

/**
   Descreve a quantidade de um artigo comprado e seu preço.
*/
public class LineItem
{
   /**
      Calcula o custo total desse item de linha.
      @return preço total
   */
   public double getTotalPrice()
   {
   }

   /**
      Formata esse item.
      @return string formatada desse item de linha
   */
   public String format()
```

```
    {
    }
}
/**
    Descreve um produto com uma descrição e um preço.
*/
public class Product
{
    /**
        Obtém a descrição do produto.
        @return descrição
    */
    public String getDescription()
    {
    }

    /**
        Obtém o preço do produto.
        @return preço unitário
    */
    public double getPrice()
    {
    }
}

/**
    Descreve um endereço.
*/
public class Address
{
    /**
        Formata o endereço.
        @return endereço como uma string com três linhas
    */
    public String format()
    {
    }
}
```

Então execute o programa `javadoc` para obter uma versão elegantemente formatada da documentação em HTML (veja Figura 8).

Essa abordagem para documentar suas classes tem várias vantagens. Você pode compartilhar a documentação em HTML com outros, se você trabalhar em equipe. Você usa um formato que é imediatamente útil – arquivos-fonte Java que você pode levar para a fase de implementação. E, acima de tudo, você fornece os comentários dos métodos-chave – uma tarefa que programadores menos preparados deixam para mais tarde e, então, muitas vezes negligenciam por falta de tempo.

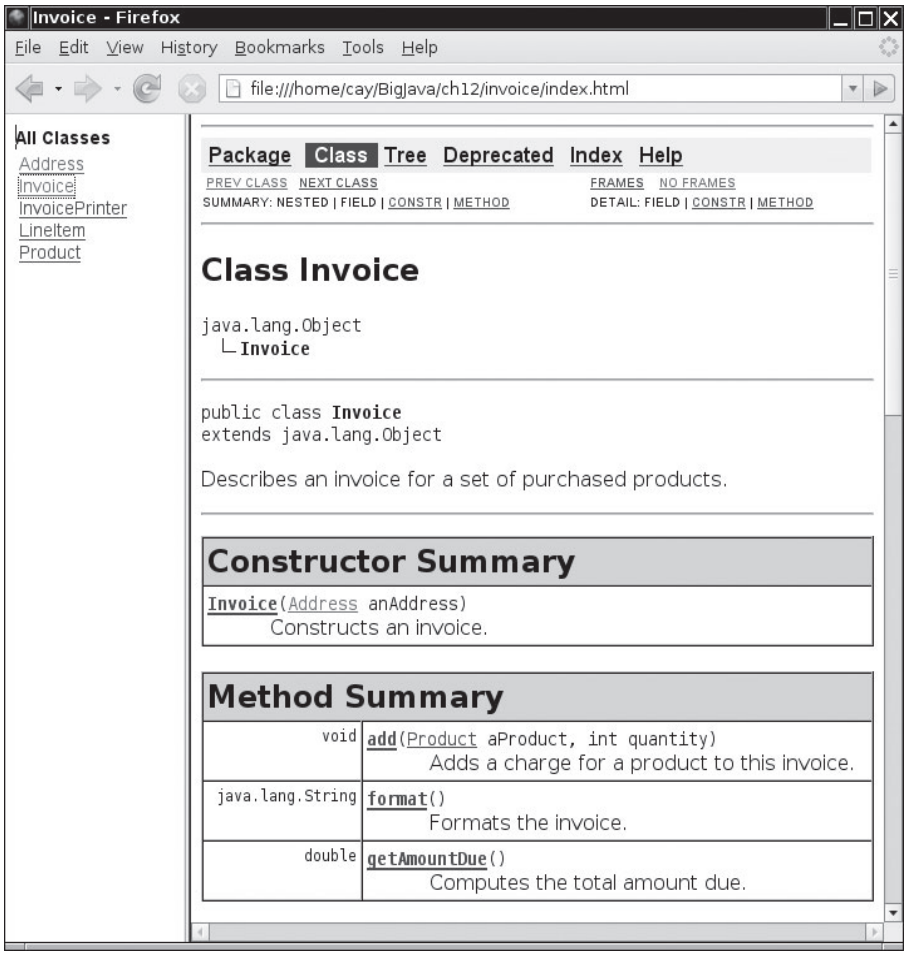


Figura 8 Documentação de classe em formato HTML.

12.4.5 Implementação

Por fim, você está pronto para implementar as classes.

Você já tem as assinaturas de método e comentários do passo anterior. Agora veja o diagrama UML para adicionar campos de instância. A agregação de classes produz campos de instância. Inicie com a classe `Invoice`. Uma fatura agrega `Address` e `LineItem`. Cada fatura tem um único endereço de cobrança, mas pode ter muitos itens de linha. Para armazenar múltiplos objetos `LineItem`, você pode utilizar uma lista de arrays. Agora você tem os campos de instância da classe `Invoice`:

```
public class Invoice
{
   . . .
   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

Um item de linha precisa armazenar um objeto `Product` e a quantidade do produto. Isso leva aos seguintes campos de instância:

```
public class LineItem
{
   . . .
   private int quantity;
   private Product theProduct;
}
```

Os métodos em si agora são muito fáceis. Eis um exemplo típico. Você já sabe o que o método `getTotalPrice` da classe `LineItem` precisa fazer – obter o preço unitário do produto e multiplicá-lo pela quantidade.

```
/**
   Calcula o custo total desse item de linha.
   @return preço total
*/
public double getTotalPrice()
{
   return theProduct.getPrice() * quantity;
}
```

Não discutiremos os outros métodos em detalhes – eles são igualmente simples e diretos.

Por fim, você precisa fornecer construtores, outra tarefa de rotina.

Eis o programa inteiro. É uma boa prática percorrê-lo detalhadamente e corresponder as classes e métodos com as fichas CRC e o diagrama UML.

ch12/invoice/InvoicePrinter.java

```
1  /**
2     Esse programa demonstra as classes de fatura
3     imprimindo uma fatura de exemplo.
4  */
5  public class InvoicePrinter
6  {
7     public static void main(String[] args)
8     {
9        Address samsAddress
10             = new Address("Sam's Small Appliances",
11             "100 Main Street", "Anytown", "CA", "98765");
12
13       Invoice samsInvoice = new Invoice(samsAddress);
14       samsInvoice.add(new Product("Toaster", 29.95), 3);
15       samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16       samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17
18       System.out.println(samsInvoice.format());
19    }
20 }
```

ch12/invoice/Invoice.java

```
1  import java.util.ArrayList;
2
3  /**
4     Descreve uma fatura para um conjunto de produtos comprados.
5  */
6  public class Invoice
7  {
8     /**
9        Constrói uma fatura.
10       @param anAddress endereço de cobrança
11    */
12    public Invoice(Address anAddress)
13    {
14       items = new ArrayList<LineItem>();
15       billingAddress = anAddress;
16    }
17
18    /**
19       Adiciona uma taxa sobre um produto a essa fatura.
20       @param aProduct produto que o cliente encomendou
21       @param quantity quantidade do produto
22    */
23    public void add(Product aProduct, int quantity)
24    {
25       LineItem anItem = new LineItem(aProduct, quantity);
26       items.add(anItem);
27    }
28
29    /**
30       Formata a fatura.
31       @return fatura formatada
32    */
33    public String format()
34    {
35       String r =  "                    I N V O I C E\n\n"
36             + billingAddress.format()
37             + String.format("\n\n%-30s%8s%5s%8s\n",
38             "Description", "Price", "Qty", "Total");
39
40       for (LineItem i : items)
41       {
42          r = r + i.format() + "\n";
43       }
44
45       r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
46
47       return r;
48    }
49
50    /**
51       Calcula o valor devido total.
52       @return valor devido
53    */
```

```
   public double getAmountDue()
   {
      double amountDue = 0;
      for (LineItem i : items)
      {
         amountDue = amountDue + i.getTotalPrice();
      }
      return amountDue;
   }

   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

ch12/invoice/LineItem.java

```
/**
   Descreve a quantidade de um artigo comprado.
*/
public class LineItem
{
   /**
      Constrói um item a partir do produto e quantidade.
      @param aProduct produto
      @param aQuantity quantidade do item
   */
   public LineItem(Product aProduct, int aQuantity)
   {
      theProduct = aProduct;
      quantity = aQuantity;
   }

   /**
      Calcula o custo total desse item de linha.
      @return preço total
   */
   public double getTotalPrice()
   {
      return theProduct.getPrice() * quantity;
   }

   /**
      Formata esse item.
      @return string formatada desse item de linha
   */
   public String format()
   {
      return String.format("%-30s%8.2f%5d%8.2f",
            theProduct.getDescription(), theProduct.getPrice(),
            quantity, getTotalPrice());
   }

   private int quantity;
   private Product theProduct;
}
```

ch12/invoice/Product.java

```
1  /**
2     Descreve um produto com uma descrição e um preço.
3  */
4  public class Product
5  {
6     /**
7        Constrói um produto a partir de uma descrição e um preço.
8        @param aDescription descrição do produto
9        @param aPrice preço do produto
10    */
11    public Product(String aDescription, double aPrice)
12    {
13       description = aDescription;
14       price = aPrice;
15    }
16
17    /**
18       Obtém a descrição do produto.
19       @return descrição
20    */
21    public String getDescription()
22    {
23       return description;
24    }
25
26    /**
27       Obtém o preço de produto.
28       @return preço unitário
29    */
30    public double getPrice()
31    {
32       return price;
33    }
34
35    private String description;
36    private double price;
37 }
```

ch12/invoice/Address.java

```
1  /**
2     Descreve um endereço.
3  */
4  public class Address
5  {
6     /**
7        Constrói um endereço.
8        @param aName nome do destinatário
9        @param aStreet rua
10       @param aCity cidade
11       @param aState código do estado em duas letras
12       @param aZip CEP
13    */
```

```
   public Address(String aName, String aStreet,
         String aCity, String aState, String aZip)
   {
      name = aName;
      street = aStreet;
      city = aCity;
      state = aState;
      zip = aZip;
   }

   /**
      Formata o endereço.
      @return endereço como uma string com três linhas
   */
   public String format()
   {
      return name + "\n" + street + "\n"
            + city + ", " + state + " " + zip;
   }

   private String name;
   private String street;
   private String city;
   private String state;
   private String zip;
}
```

Autoverificação da aprendizagem

10. Qual classe é responsável pelo cálculo do valor devido? Quais são seus colaboradores nessa tarefa?
11. Por que os métodos `format` retornam objetos `String` em vez de imprimir diretamente para `System.out`?

12.5 Estudo de caso: Um caixa automático

12.5.1 Requisitos

O propósito deste projeto é planejar a simulação de um caixa automático (*Automatic Teller Machine* – ATM). O ATM é utilizado pelos clientes de um banco. Cada cliente tem duas contas: uma conta corrente e uma conta poupança. Cada cliente também tem um número de cliente e um número de identificação pessoal (*Personal Identification Number* – PIN); ambos são necessários para obter acesso às contas. (Em um ATM real, o número do cliente seria registrado na fita magnética do cartão bancário de ATM. Nessa simulação, o cliente precisará digitá-lo.) Com o ATM, os clientes podem selecionar uma conta (corrente ou poupança). O saldo da conta selecionada é exibido. Então o cliente pode depositar e retirar dinheiro. Esse processo é repetido até que o cliente escolha sair.

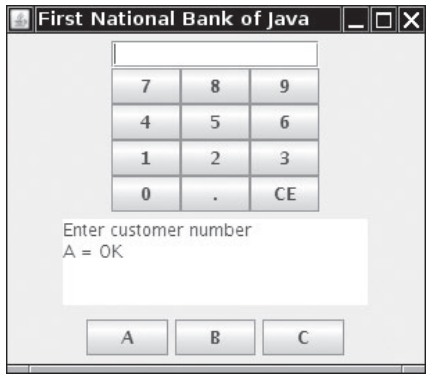


Figura 9
Interface gráfica com o usuário (GUI) do ATM.

Os detalhes da interação do usuário dependem da interface com o usuário que escolhemos para a simulação. Desenvolveremos duas interfaces separadas: uma interface gráfica que simula rigorosamente um ATM real (veja Figura 9) e uma interface baseada em texto que permite testar o ATM e classes de banco sem sermos distraídos pela programação da GUI.

Na interface GUI, o ATM tem um teclado para inserir números, uma tela para exibir mensagens e um conjunto de botões, rotulados A, B e C, cuja função depende do estado da máquina.

Especificamente, a interação do usuário ocorre da seguinte maneira. Depois de iniciado, o ATM espera que um usuário insira um número de cliente. A tela mostra a seguinte mensagem:

```
Enter customer number
A = OK
```

O usuário insere o número do cliente no teclado e pressiona o botão A. A mensagem da tela muda para:

```
Enter PIN
A = OK
```

Em seguida, o usuário insere o PIN e pressiona o botão A novamente. Se o número e o identificador (ID) de cliente corresponderem com os de um dos clientes no banco, então o cliente pode prosseguir. Se não, o usuário é novamente solicitado a inserir o número de cliente.

Se o cliente for autorizado para utilizar o sistema, a mensagem da tela muda para:

```
Select Account
A = Checking
B = Savings
C = Exit
```

Se o usuário pressionar o botão C, o ATM volta ao seu estado original e pede ao próximo usuário para inserir um número de cliente.

Se o usuário pressionar os botões A ou B, o ATM se lembrará da conta selecionada e a mensagem da tela mudará para:

```
Balance = o saldo da conta selecionada
Enter amount and select transaction
A = Withdraw
```

```
B = Deposit
C = Cancel
```

Se o usuário pressionar os botões A ou B, o valor inserido no teclado é retirado da conta selecionada, ou depositado nela. (Isso é só uma simulação, então nenhum dinheiro é liberado e nenhum depósito é aceito.) Depois, o ATM volta ao estado anterior, permitindo ao usuário selecionar outra conta ou sair.

Se o usuário pressionar o botão C, o caixa volta para o estado precedente sem executar qualquer transação.

Na interação baseada em texto, lemos a entrada a partir de `System.in` em vez dos botões. Eis um típico diálogo:

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```

Em nossa solução, somente as classes de interface com o usuário são afetadas pela escolha de interface com o usuário. O restante das classes pode ser utilizado para ambas as soluções – elas são desacopladas da interface com o usuário.

Como essa é uma simulação, o ATM não se comunica realmente com um banco. Ele simplesmente carrega um conjunto de números de cliente e PINs de um arquivo. Todas as contas são inicializadas com saldo zero.

12.5.2 Fichas CRC

Seguiremos novamente a receita da Seção 12.2 e mostraremos como descobrir classes, responsabilidades e relacionamentos, e como obter um projeto detalhado para o programa de ATM.

Lembre-se de que a primeira regra para localizar classes é "Procurar substantivos na descrição do problema". Eis uma lista dos substantivos:

```
ATM (Caixa automático)
User (Usuário)
Keypad (Teclado)
Display (Tela)
Display message (Mensagem da tela)
Button (Botão)
State (Estado)
Bank account (Conta bancária)
Checking account (Conta corrente)
Savings account (Conta poupança)
Customer (Cliente)
Customer number (Número de cliente)
PIN (Número de identificação pessoal)
Bank (Banco)
```

Naturalmente, nem todos esses substantivos se tornarão nomes de classes e ainda podemos descobrir a necessidade de classes que não estão nessa lista, mas esse é um bom começo.

Os usuários e clientes representam o mesmo conceito nesse programa. Vamos utilizar uma classe `Customer`. Um cliente tem duas contas bancárias e exigiremos que um objeto `Customer` deva ser capaz de localizar essas contas. (Outro possível projeto tornaria a classe `Bank` responsável por localizar as contas de um dado cliente – veja o Exercício P12.9.)

Um cliente também possui um número de cliente e um PIN. Podemos, naturalmente, exigir que um objeto cliente nos dê o número do cliente e o PIN. Mas talvez isso não seja tão seguro. Em vez disso, simplesmente exija que o objeto cliente, quando lhe for dado um número de cliente e um PIN, nos diga se eles correspondem a suas próprias informações ou não.

Customer
obter contas
corresponder número e PIN

Um banco contém uma coleção de clientes. Quando um usuário usa um ATM e digita seu número de cliente e PIN, é trabalho do banco localizar o cliente correspondente. Como o banco pode fazer isso? Ele precisa verificar, para cada cliente, se o número de cliente e o PIN correspondem. Portanto, ele precisa chamar o método *corresponder número e PIN* da classe `Customer` que acabamos de descobrir. Como o método *localizar cliente* chama um método `Customer`, ele colabora com a classe `Customer`. Registramos esse fato na coluna direita da ficha CRC.

Quando a simulação iniciar, o banco também deve saber ler informações de conta a partir de um arquivo.

Bank	
localizar cliente	`Customer`
ler clientes	

A classe `BankAccount` já é nossa conhecida com métodos para obter o saldo e para depositar e retirar dinheiro.

Nesse programa não há nada que distinga contas corrente de contas poupança. O ATM não paga juros nem cobra taxas. Portanto, decidimos não implementar subclasses separadas para conta corrente e conta poupança.

Por fim, sobra-nos a própria classe ATM. Uma noção importante do ATM é o estado. O estado atual da máquina determina o texto dos prompts e a função dos botões. Por exemplo, ao efetuar logon pela primeira vez, você usa os botões A e B para selecionar uma conta. Em seguida, você usa os mesmos botões para escolher entre depósito e retirada. O ATM deve lembrar do estado atual para interpretar corretamente os botões.

Há quatro estados:

1. START: Insira ID do cliente
2. PIN: Insira senha PIN
3. ACCOUNT: Selecione conta
4. TRANSACT: Selecione transação

Para entender como mudar de um estado para o seguinte, é útil elaborar um diagrama de estados (Figura 10). A notação UML tem formas padronizadas para diagramas de estado. Desenhe estados como retângulos com cantos arredondados. Desenhe as alterações de estado como setas, com rótulos que indicam a razão da alteração.

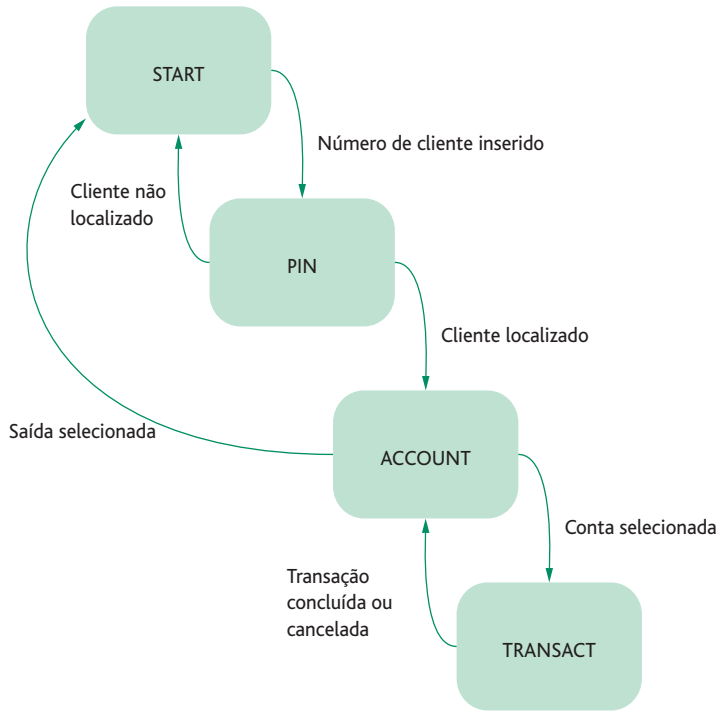


Figura 10 Diagrama para a classe ATM de estados.

O usuário deve digitar um número válido de cliente e o PIN. Então o ATM pode solicitar ao banco para localizar o cliente. Isso exige um método *selecionar cliente*. Ele colabora com o banco, perguntando a ele sobre o cliente que corresponde ao número de cliente e PIN. Em seguida, há um método *selecionar conta* que pede para o cliente atual a conta corrente ou conta poupança. Por fim, o ATM deve executar a transação selecionada na conta atual.

ATM	
gerenciar estados	`Customer`
selecionar cliente	`Bank`
selecionar conta	`BankAccount`
executar transação	

Naturalmente, descobrir essas classes e métodos não foi tão elegante e ordenado como parece a partir dessa discussão. Quando projetei essas classes para este livro, fiz várias tentativas e rasguei muitas fichas para chegar a um projeto satisfatório. Também é importante lembrar-se de que raramente há o melhor projeto.

Esse projeto tem várias vantagens. As classes descrevem conceitos claros. Os métodos são suficientes para implementar todas as tarefas necessárias. (Repassei mentalmente cada possível cenário de uso do ATM para verificar isso.) Não há uma quantidade exagerada de dependências de colaboração entre as classes. Portanto, fiquei satisfeito com esse projeto e prossegui ao próximo passo.

12.5.3 Diagramas UML

A Figura 11 mostra os relacionamentos entre essas classes, utilizando a interface gráfica com o usuário. (A interface com o usuário do console usa uma única classe `ATMSimulator` em vez das classes `ATMFrame` e `Keypad`.)

Para elaborar as dependências, utilize as colunas "colaborador" das fichas CRC. Examinando essas colunas, você vê que as dependências são as seguintes:

- `ATM` usa `Bank`, `Customer` e `BankAccount`.
- `Bank` usa `Customer`.
- `Customer` usa `BankAccount`.

É fácil ver alguns relacionamentos de agregação. Um banco tem clientes e cada cliente tem duas contas bancárias.

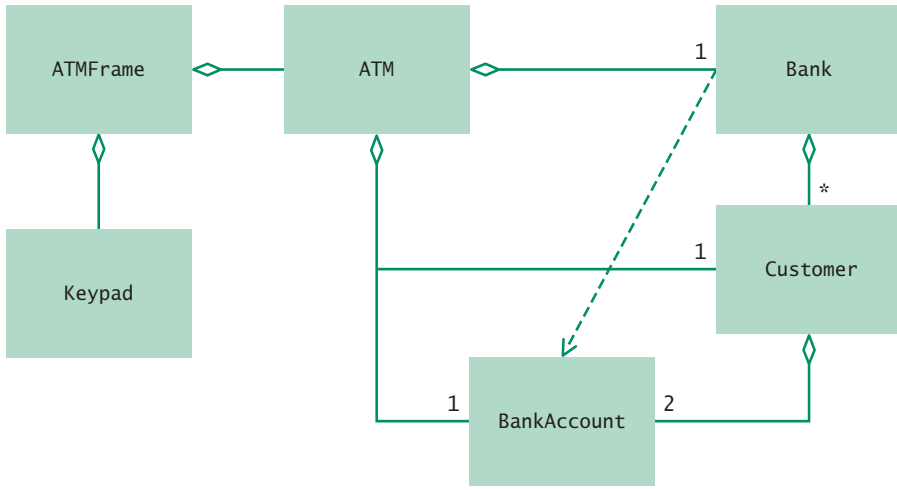


Figura 11 Relacionamentos entre as classes do ATM.

A classe `ATM` agrega `Bank`? Para responder a essa pergunta, pergunte a você mesmo se um objeto ATM precisa armazenar uma referência a um objeto banco. Ele precisa localizar o mesmo objeto banco entre diversas chamadas de método? De fato, ele precisa. Portanto, a agregação é o relacionamento apropriado.

Um ATM agrega clientes? Claramente, ele não é responsável por armazenar todos os clientes do banco. Isso é trabalho do banco. Mas no nosso projeto, o ATM lembra-se do cliente *atual*. Se um cliente fez login, os comandos subseqüentes referem-se ao mesmo cliente. O ATM precisa armazenar uma referência ao cliente ou pedir ao banco para pesquisar o objeto sempre que precisar do cliente atual. É uma decisão de projeto: armazenar o objeto ou pesquisá-lo quando necessário. Decidiremos armazenar o objeto cliente atual. Isto é, usaremos agregação. Observe que a escolha da agregação não é uma conseqüência automática da descrição do problema – é uma decisão de projeto.

Da mesma forma, decidiremos armazenar a conta bancária atual (corrente ou poupança) que o usuário seleciona. Portanto, temos um relacionamento de agregação entre `ATM` e `BankAccount`.

O diagrama de classes é uma boa ferramenta para visualizar dependências. Examine as classes GUI. Elas são completamente independentes do restante do sistema ATM. Você pode substituir a GUI por uma interface de console e pode tirar a classe `Keypad` e usá-la em outro aplicativo. Além disso, as classes `Bank`, `BankAccount` e `Customer`, embora dependentes uma da outra, não sabem nada sobre a classe `ATM`. Isso faz sentido – você pode ter bancos sem caixas eletrônicos. Como pode ver, ao analisar relacionamentos, procuramos tanto a ausência como a presença de relacionamentos.

12.5.4 Documentação de métodos

Agora você está pronto para o passo final da fase de projeto: documentar as classes e métodos que descobriu. Eis uma parte da documentação da classe `ATM`:

```
/**
   Um ATM que acessa um banco.
*/
public class ATM
{
   /**
      Constrói um ATM para um dado banco.
      @param aBank banco ao qual esse ATM se conecta
   */
   public ATM(Bank aBank) { }

   /**
      Configura o número do cliente atual
      e determina o estado para o PIN.
      (Pré-condição: o estado é START)
      @param number número do cliente
   */
   public void setCustomerNumber(int number) { }

   /**
      Localiza o cliente do banco.
      Se localizado, define o estado como ACCOUNT, senão START.
      (Pré-condição: o estado é PIN)
      @param pin PIN do cliente atual
   */
   public void selectCustomer(int pin) { }

   /**
      Configura a conta atual como corrente ou poupança. Configura
      estado como TRANSACT.
      (Pré-condição: o estado é ACCOUNT ou TRANSACT)
      @param account CHECKING ou SAVINGS
   */
   public void selectAccount(int account) { }

   /**
      Retira o valor da conta atual.
      (Pré-condição: o estado é TRANSACT)
      @param value valor a retirar
   */
   public void withdraw(double value) { }
   . . .
}
```

Agora execute o utilitário `javadoc` para colocar essa documentação no formato HTML.

Por concisão, omitimos a documentação das outras classes.

12.5.5 Implementação

Por fim, chegou a hora de implementar o simulador de ATM. A fase de implementação é muito simples e direta e deve durar *muito menos que a fase de projeto.*

Uma boa estratégia para implementar as classes é ir de "baixo para cima". Inicie com as classes que não dependem de outras, como `Keypad` e `BankAccount`. Então implemente uma classe como `Customer` que depende somente da classe `BankAccount`. Essa abordagem "de baixo para cima" (*bottom-up*) permite testar suas classes individualmente. Você localizará as implementações dessas classes no fim desta seção.

A classe mais complexa é a classe `ATM`. Para implementar os métodos, você precisa definir as variáveis de instância necessárias. A partir do diagrama de classes, você pode dizer que o ATM tem um objeto banco. Ele se torna uma variável de instância da classe:

```
public class ATM
{
   . . .
   private Bank theBank;
}
```

A partir da descrição dos estados do ATM, fica claro que solicitamos variáveis de instância adicionais para armazenar o estado atual, o cliente e a conta bancária.

```
public class ATM
{  . . .
   private int state;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   . . .
}
```

A implementação da maioria dos métodos é muito simples e direta. Considere o método `selectCustomer`. A partir da documentação de projeto, temos a descrição:

```
/**
   Localiza o cliente do banco.
   Se localizado, define o estado como ACCOUNT, senão START.
   (Pré-condição: o estado é PIN)
   @param pin PIN do cliente atual
*/
```

Essa descrição pode ser traduzida quase literalmente em instruções Java:

```
public void selectCustomer(int pin)
{
   assert state == PIN;
   currentCustomer = theBank.findCustomer(customerNumber, pin);
   if (currentCustomer == null)
      state = START;
   else
      state = ACCOUNT;
}
```

Não vamos fazer uma descrição passo a passo de cada método do programa ATM. Você deve reservar um tempo e comparar a implementação real com as fichas CRC e o diagrama UML.

ch12/atm/ATM.java

```
/**
   Um ATM que acessa um banco.
*/
public class ATM
{
   /**
      Constrói um ATM para um dado banco.
      @param aBank banco ao qual esse ATM se conecta
   */
   public ATM(Bank aBank)
   {
      theBank = aBank;
      reset();
   }

   /**
      Redefine o ATM para o estado inicial.
   */
   public void reset()
   {
      customerNumber = -1;
      currentAccount = null;
      state = START;
   }

   /**
      Configura o número do cliente atual
      e determina o estado para o PIN.
      (Pré-condição: o estado é START)
      @param number número do cliente
   */
   public void setCustomerNumber(int number)
   {
      assert state == START;
      customerNumber = number;
      state = PIN;
   }

   /**
      Localiza o cliente do banco.
      Se localizado, define o estado como ACCOUNT, caso contrário START.
      (Pré-condição: o estado é PIN)
      @param pin PIN do cliente atual
   */
   public void selectCustomer(int pin)
   {
      assert state == PIN;
      currentCustomer = theBank.findCustomer(customerNumber, pin);
      if (currentCustomer == null)
         state = START;
      else
         state = ACCOUNT;
   }
```

```
54
55      /**
56         Configura a conta atual como corrente ou poupança. Configura
57         estado como TRANSACT.
58         (Pré-condição: o estado é ACCOUNT ou TRANSACT)
59         @param account CHECKING ou SAVINGS
60      */
61      public void selectAccount(int account)
62      {
63         assert state == ACCOUNT || state == TRANSACT;
64         if (account == CHECKING)
65            currentAccount = currentCustomer.getCheckingAccount();
66         else
67            currentAccount = currentCustomer.getSavingsAccount();
68         state = TRANSACT;
69      }
70
71      /**
72         Retira o valor da conta atual.
73         (Pré-condição: o estado é TRANSACT)
74         @param value valor a retirar
75      */
76      public void withdraw(double value)
77      {
78         assert state == TRANSACT;
79         currentAccount.withdraw(value);
80      }
81
82      /**
83         Deposita o valor na conta atual.
84         (Pré-condição: o estado é TRANSACT)
85         @param value valor a depositar
86      */
87      public void deposit(double value)
88      {
89         assert state == TRANSACT;
90         currentAccount.deposit(value);
91      }
92
93      /**
94         Obtém o saldo da conta atual.
95         (Pré-condição: o estado é TRANSACT)
96         @return saldo
97      */
98      public double getBalance()
99      {
100        assert state == TRANSACT;
101        return currentAccount.getBalance();
102     }
103
104     /**
105        Volta para o estado anterior.
106     */
```

```
   public void back()
   {
      if (state == TRANSACT)
         state = ACCOUNT;
      else if (state == ACCOUNT)
         state = PIN;
      else if (state == PIN)
         state = START;
   }

   /**
      Obtém o estado atual desse ATM.
      @return estado atual
   */
   public int getState()
   {
      return state;
   }

   private int state;
   private int customerNumber;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   private Bank theBank;

   public static final int START = 1;
   public static final int PIN = 2;
   public static final int ACCOUNT = 3;
   public static final int TRANSACT = 4;

   public static final int CHECKING = 1;
   public static final int SAVINGS = 2;
}
```

ch12/atm/Bank.java

```
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
   Um banco contém clientes com contas bancárias.
*/
public class Bank
{
   /**
      Constrói um banco sem clientes.
   */
```

```
14      public Bank()
15      {
16         customers = new ArrayList<Customer>();
17      }
18
19      /**
20         Lê os números de cliente e PINs
21         e inicializa as contas bancárias.
22         @param filename nome do arquivo do cliente
23      */
24      public void readCustomers(String filename)
25            throws IOException
26      {
27         Scanner in = new Scanner(new FileReader(filename));
28         while (in.hasNext())
29         {
30            int number = in.nextInt();
31            int pin = in.nextInt();
32            Customer c = new Customer(number, pin);
33            addCustomer(c);
34         }
35         in.close();
36      }
37
38      /**
39         Adiciona um cliente ao banco.
40         @param c cliente a adicionar
41      */
42      public void addCustomer(Customer c)
43      {
44         customers.add(c);
45      }
46
47      /**
48         Localiza um cliente no banco.
49         @param aNumber número do cliente
50         @param aPin número da identificação pessoal (senha)
51         @return cliente correspondente, ou null se nenhum cliente
52         corresponde
53      */
54      public Customer findCustomer(int aNumber, int aPin)
55      {
56         for (Customer c : customers)
57         {
58            if (c.match(aNumber, aPin))
59               return c;
60         }
61         return null;
62      }
63
64      private ArrayList<Customer> customers;
65   }
```

ch12/atm/Customer.java

```
/**
   Um cliente de banco com uma conta corrente e uma conta poupança.
*/
public class Customer
{
   /**
      Constrói um cliente com um número dado e um PIN.
      @param aNumber número do cliente
      @param aPin número de identificação pessoal (senha)
   */
   public Customer(int aNumber, int aPin)
   {
      customerNumber = aNumber;
      pin = aPin;
      checkingAccount = new BankAccount();
      savingsAccount = new BankAccount();
   }

   /**
      Testa se esse cliente corresponde a um número de cliente
      e PIN.
      @param aNumber número de cliente
      @param aPin número de identificação pessoal
      @return true se o número de cliente e PIN correspondem
   */
   public boolean match(int aNumber, int aPin)
   {
      return customerNumber == aNumber && pin == aPin;
   }

   /**
      Obtém a conta bancária desse cliente.
      @return conta corrente
   */
   public BankAccount getCheckingAccount()
   {
      return checkingAccount;
   }

   /**
      Obtém a conta poupança desse cliente.
      @return conta corrente
   */
   public BankAccount getSavingsAccount()
   {
      return savingsAccount;
   }

   private int customerNumber;
   private int pin;
   private BankAccount checkingAccount;
   private BankAccount savingsAccount;
}
```

A seguinte classe implementa uma interface com o usuário de console para o ATM.

ch12/atm/ATMSimulator.java

```
import java.io.IOException;
import java.util.Scanner;

/**
   Uma simulação baseada em texto de um ATM.
*/
public class ATMSimulator
{
   public static void main(String[] args)
   {
      ATM theATM;
      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch(IOException e)
      {
         System.out.println("Error opening accounts file.");
         return;
      }

      Scanner in = new Scanner(System.in);

      while (true)
      {
         int state = theATM.getState();
         if (state == ATM.START)
         {
            System.out.print("Enter customer number: ");
            int number = in.nextInt();
            theATM.setCustomerNumber(number);
         }
         else if (state == ATM.PIN)
         {
            System.out.print("Enter PIN: ");
            int pin = in.nextInt();
            theATM.selectCustomer(pin);
         }
         else if (state == ATM.ACCOUNT)
         {
            System.out.print("A=Checking, B=Savings, C=Quit: ");
            String command = in.next();
            if (command.equalsIgnoreCase("A"))
               theATM.selectAccount(ATM.CHECKING);
            else if (command.equalsIgnoreCase("B"))
               theATM.selectAccount(ATM.SAVINGS);
            else if (command.equalsIgnoreCase("C"))
               theATM.reset();
```

```
            else
                System.out.println("Illegal input!");
         }
         else if (state == ATM.TRANSACT)
         {
            System.out.println("Balance=" + theATM.getBalance());
            System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
            String command = in.next();
            if (command.equalsIgnoreCase("A"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.deposit(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("B"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.withdraw(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("C"))
               theATM.back();
            else
               System.out.println("Illegal input!");
         }
      }
   }
}
```

Saída

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
. . .
```

Eis as classes de interface com o usuário da versão GUI da interface com o usuário.

ch12/atm/ATMViewer.java

```
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/**
   Uma simulação gráfica de um ATM.
*/
```

```
8  public class ATMViewer
9  {
10    public static void main(String[] args)
11    {
12       ATM theATM;
13
14       try
15       {
16          Bank theBank = new Bank();
17          theBank.readCustomers("customers.txt");
18          theATM = new ATM(theBank);
19       }
20       catch(IOException e)
21       {
22          JOptionPane.showMessageDialog(null,
23                "Error opening accounts file.");
24          return;
25       }
26
27       JFrame frame = new ATMFrame(theATM);
28       frame.setTitle("First National Bank of Java");
29       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30       frame.setVisible(true);
31    }
32 }
```

ch12/atm/ATMFrame.java

```
1  import java.awt.FlowLayout;
2  import java.awt.GridLayout;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import javax.swing.JButton;
6  import javax.swing.JFrame;
7  import javax.swing.JPanel;
8  import javax.swing.JTextArea;
9
10 /**
11    Um frame que exibe os componentes de um ATM.
12 */
13 public class ATMFrame extends JFrame
14 {
15    /**
16       Constrói a interface com o usuário do frame de ATM.
17    */
18    public ATMFrame(ATM anATM)
19    {
20       theATM = anATM;
21
22       // Constrói componentes
23       pad = new KeyPad();
24
25       display = new JTextArea(4, 20);
26
```

```
27         aButton = new JButton("  A  ");
28         aButton.addActionListener(new AButtonListener());
29
30         bButton = new JButton("  B  ");
31         bButton.addActionListener(new BButtonListener());
32
33         cButton = new JButton("  C  ");
34         cButton.addActionListener(new CButtonListener());
35
36         // Adiciona componentes
37
38         JPanel buttonPanel = new JPanel();
39         buttonPanel.add(aButton);
40         buttonPanel.add(bButton);
41         buttonPanel.add(cButton);
42
43         setLayout(new FlowLayout());
44         add(pad);
45         add(display);
46         add(buttonPanel);
47         showState();
48
49         setSize(FRAME_WIDTH, FRAME_HEIGHT);
50      }
51
52      /**
53         Atualiza a mensagem exibida.
54      */
55      public void showState()
56      {
57         int state = theATM.getState();
58         pad.clear();
59         if (state == ATM.START)
60            display.setText("Enter customer number\nA = OK");
61         else if (state == ATM.PIN)
62            display.setText("Enter PIN\nA = OK");
63         else if (state == ATM.ACCOUNT)
64            display.setText("Select Account\n"
65                  + "A = Checking\nB = Savings\nC = Exit");
66         else if (state == ATM.TRANSACT)
67            display.setText("Balance = "
68                  + theATM.getBalance()
69                  + "\nEnter amount and select transaction\n"
70                  + "A = Withdraw\nB = Deposit\nC = Cancel");
71      }
72
73      private class AButtonListener implements ActionListener
74      {
75         public void actionPerformed(ActionEvent event)
76         {
77            int state = theATM.getState();
78            if (state == ATM.START)
79               theATM.setCustomerNumber((int) pad.getValue());
80            else if (state == ATM.PIN)
81               theATM.selectCustomer((int) pad.getValue());
```

```
82          else if (state == ATM.ACCOUNT)
83             theATM.selectAccount(ATM.CHECKING);
84          else if (state == ATM.TRANSACT)
85          {
86             theATM.withdraw(pad.getValue());
87             theATM.back();
88          }
89          showState();
90       }
91    }
92
93    private class BButtonListener implements ActionListener
94    {
95       public void actionPerformed(ActionEvent event)
96       {
97          int state = theATM.getState();
98          if (state == ATM.ACCOUNT)
99             theATM.selectAccount(ATM.SAVINGS);
100         else if (state == ATM.TRANSACT)
101         {
102            theATM.deposit(pad.getValue());
103            theATM.back();
104         }
105         showState();
106      }
107   }
108
109   private class CButtonListener implements ActionListener
110   {
111      public void actionPerformed(ActionEvent event)
112      {
113         int state = theATM.getState();
114         if (state == ATM.ACCOUNT)
115            theATM.reset();
116         else if (state == ATM.TRANSACT)
117            theATM.back();
118         showState();
119      }
120   }
121
122   private JButton aButton;
123   private JButton bButton;
124   private JButton cButton;
125
126   private KeyPad pad;
127   private JTextArea display;
128
129   private ATM theATM;
130
131   private static final int FRAME_WIDTH = 300;
132   private static final int FRAME_HEIGHT = 300;
133 }
```

Essa classe usa gerenciadores de leiaute para organizar o campo de texto e os botões de teclado. Veja o Capítulo 18 para obter mais informações sobre os gerenciadores de leiaute.

ch12/atm/KeyPad.java

```
1  import java.awt.BorderLayout;
2  import java.awt.GridLayout;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import javax.swing.JButton;
6  import javax.swing.JPanel;
7  import javax.swing.JTextField;
8
9  /**
10    Um componente que deixa o usuário inserir um número, usando
11    um teclado rotulado com dígitos.
12 */
13 public class KeyPad extends JPanel
14 {
15    /**
16       Constrói o painel do teclado.
17    */
18    public KeyPad()
19    {
20       setLayout(new BorderLayout());
21
22       // Adiciona o campo de exibição
23
24       display = new JTextField();
25       add(display, "North");
26
27       // Faz o painel dos botões
28
29       buttonPanel = new JPanel();
30       buttonPanel.setLayout(new GridLayout(4, 3));
31
32       // Adiciona botões de dígito
33
34       addButton("7");
35       addButton("8");
36       addButton("9");
37       addButton("4");
38       addButton("5");
39       addButton("6");
40       addButton("1");
41       addButton("2");
42       addButton("3");
43       addButton("0");
44       addButton(".");
45
46       // Adiciona botão para limpar a entrada
47
48       clearButton = new JButton("CE");
```

```
49         buttonPanel.add(clearButton);
50
51         class ClearButtonListener implements ActionListener
52         {
53            public void actionPerformed(ActionEvent event)
54            {
55               display.setText("");
56            }
57         }
58         ActionListener listener = new ClearButtonListener();
59
60         clearButton.addActionListener(new
61               ClearButtonListener());
62
63         add(buttonPanel, "Center");
64      }
65
66      /**
67         Adiciona um botão ao painel de botões.
68         @param label rótulo do botão
69      */
70      private void addButton(final String label)
71      {
72         class DigitButtonListener implements ActionListener
73         {
74            public void actionPerformed(ActionEvent event)
75            {
76
77               // Não adiciona dois pontos de fração decimal
78               if (label.equals(".")
79                     && display.getText().indexOf(".") != -1)
80                  return;
81
82               // Anexa texto de rótulo ao botão
83               display.setText(display.getText() + label);
84            }
85         }
86
87         JButton button = new JButton(label);
88         buttonPanel.add(button);
89         ActionListener listener = new DigitButtonListener();
90         button.addActionListener(listener);
91      }
92
93      /**
94         Obtém o valor que o usuário inseriu.
95         @return valor no campo de texto do teclado
96      */
97      public double getValue()
98      {
99         return Double.parseDouble(display.getText());
100     }
101
```

```
102     /**
103        Limpa a tela.
104     */
105     public void clear()
106     {
107        display.setText("");
108     }
109
110     private JPanel buttonPanel;
111     private JButton clearButton;
112     private JTextField display;
113  }
```

Neste capítulo, você aprendeu uma abordagem sistemática para construir um programa relativamente complexo. Entretanto, o projeto orientado a objetos não é definitivamente um esporte de grande apelo popular. Para realmente aprender a projetar e implementar programas, você precisa ganhar experiência repetindo esse processo com seus próprios projetos. É bem possível que você não obtenha de imediato uma boa solução e que precise voltar e reorganizar suas classes e responsabilidades. Isso é normal e mais do que esperado. O propósito do processo de projeto orientado a objetos é indicar precisamente esses problemas na fase de projeto, quando eles ainda são fáceis de corrigir, em vez de na fase de implementação, quando a reorganização maciça é mais difícil e demorada.

Autoverificação da aprendizagem

12. Por que a classe `Bank` nesse exemplo não armazena uma lista de array de contas bancárias?
13. Suponha a mudança de requisitos – você precisa salvar os saldos da conta atual em um arquivo depois de cada transação e recarregá-los quando o programa iniciar. Qual é o impacto dessa alteração no projeto?

Fato aleatório 12.2

Desenvolvimento de software – arte ou ciência?
O Fato Aleatório 12.2 discute se os desenvolvedores de software são mais bem caracterizados como artistas, artesões, cientistas ou engenheiros.

RESUMO DO CAPÍTULO

1. O ciclo de vida de software engloba todas as atividades, da análise inicial até a obsolescência.
2. Um processo formal para desenvolvimento de software descreve fases do processo de desenvolvimento e dá diretrizes sobre como executá-las.

3. O modelo de desenvolvimento de software em cascata descreve um processo seqüencial de análise, projeto, implementação, teste e implantação.
4. O modelo espiral de desenvolvimento de software descreve um processo iterativo em que projeto e implementação são repetidos.
5. A programação extrema (eXtreme Programming – XP) é uma metodologia de desenvolvimento que busca obter simplicidade eliminando a estrutura formal e concentrando-se nas melhores práticas.
6. No projeto orientado a objetos, você descobre classes, determina as responsabilidades delas e descreve os relacionamentos entre elas.
7. Um cartão CRC descreve uma classe, suas responsabilidades e suas classes colaboradoras.
8. A herança (o relacionamento *é-um*) às vezes é utilizada inadequadamente quando o relacionamento *tem-um* seria mais apropriado.
9. A agregação (o relacionamento *tem-um*) denota que objetos de uma classe contêm referências a objetos de outra classe.
10. A dependência é outro nome para o relacionamento "usa".
11. Você precisa ser capaz de distinguir as notações UML para herança, implementação de interface, agregação e dependência.
12. Utilize comentários no estilo `javadoc` (com o corpo dos métodos deixados em branco) para registrar o comportamento das classes.

LEITURA ADICIONAL

1. Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley, 2005, 1999.
2. Kent Beck, *Extreme Programming Explained,* Addison-Wesley, 1999.*

EXERCÍCIOS DE REVISÃO

★ **Exercício R12.1.** O que é o ciclo de vida do software?

★★ **Exercício R12.2.** Liste os passos do processo de projeto orientado a objetos que este capítulo recomenda que o estudante use.

★ **Exercício R12.3.** Forneça uma regra geral para descobrir classes durante a fase de projeto de um programa.

★ **Exercício R12.4.** Forneça uma regra geral para descobrir métodos ao projetar um programa.

* N. de R.: Esta obra foi publicada pela Bookman Editora sob o título *Programação Extrema Explicada*.

★★ **Exercício R12.5.** Depois de descobrir um método, por que é importante identificar o objeto *responsável* por executar a ação?

★ **Exercício R12.6.** Que relacionamento é apropriado entre as seguintes classes: agregação, herança ou nenhum deles?

a. `University-Student (Universidade-Estudante)`
b. `Student-TeachingAssistant (Estudante-AssistenteDeEnsino)`
c. `Student-Freshman (Estudante-Calouro)`
d. `Student-Professor (Estudante-Professor)`
e. `Car-Door (Carro-Porta)`
f. `Truck-Vehicle (Caminhão-Veículo)`
g. `Traffic-TrafficSign (Tráfego-SinalDeTrânsito)`
h. `TrafficSign-Color (SinalDeTrânsito-Cor)`

★★ **Exercício R12.7.** Todo BMW é um veículo. Uma classe `BMW` deve herdar da classe `Vehicle`? BMW é um fabricante de veículos. Isso significa que a classe `BMW` deve herdar da classe `VehicleManufacturer`?

★★ **Exercício R12.8.** Alguns livros sobre programação orientada a objetos recomendam o uso de herança para que a classe `Circle` estenda a classe `Point`. Então a classe `Circle` herda o método `setLocation` da superclasse `Point`. Explique por que o método `setLocation` não precisa ser redefinido na subclasse. Mas por que não é uma boa idéia fazer `Circle` herdar de `Point`? Inversamente, `Point` herdaria de `Circle` para seguir a regra *é-um*? Seria uma boa idéia?

★ **Exercício R12.9.** Escreva fichas CRC para as classes `Coin` e `CashRegister` descritas na Seção 8.2.

★ **Exercício R12.10.** Escreva as fichas CRC para as classes `Bank` e `BankAccount` na Seção 7.2.

★★ **Exercício R12.11.** Desenhe um diagrama UML para as classes `Coin` e `CashRegister` descritas na Seção 8.2.

★★★ **Exercício R12.12.** Um arquivo contém um conjunto de registros que descrevem países. Cada registro consiste no nome do país, sua população e sua área. Suponha que sua tarefa seja escrever um programa que leia tal arquivo e imprima:

- o país com a maior área
- o país com a maior população
- o país com a maior densidade populacional (pessoas por quilômetro quadrado)

Reflita sobre os problemas que precisa resolver. Quais classes e métodos você precisará? Produza uma série de fichas CRC, um diagrama UML e uma série de comentários `javadoc`.

★★★ **Exercício R12.13.** Descubra classes e métodos para gerar um cartão de relatório escolar que lista todas as classes, notas e a média de um semestre. Produza uma série de fichas CRC, um diagrama UML e uma série de comentários `javadoc`.

★★★ **Exercício R12.14.** Considere um sistema de notas que avalia as respostas do aluno a questionários. Um questionário consiste em perguntas. Há diferentes tipos de perguntas, inclusive perguntas dissertativas e perguntas de múltipla escolha. Os alunos enviam os questionários e o sistema de notas os avalia. Elabore um diagrama UML para as classes `Quiz`, `Question`, `EssayQuestion`, `MultipleChoiceQuestion`, `Student` e `Submission`.

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★★ **Exercício P12.1.** Aprimore o programa de impressão de fatura para oferecer dois tipos de itens de linha: Um tipo descreve produtos que são comprados em quantidades numéricas (como "3 torradeiras") e o outro descreve uma taxa fixa (como "remessa: $5.00"). *Dica:* Utilize herança. Produza um diagrama UML da sua implementação modificada.

★★ **Exercício P12.2.** O programa de impressão de fatura é um tanto quanto irreal porque a formatação dos objetos `LineItem` não dará bons resultados visuais quando os preços e as quantidades tiverem número variável de dígitos. Aperfeiçoe o método `format` de duas maneiras: Aceite um array `int[]` de larguras de coluna como um parâmetro. Utilize a classe `NumberFormat` para formatar os valores de moeda.

★★ **Exercício P12.3.** O programa de impressão de fatura tem um defeito horrível – ele mistura a "lógica do negócio", o cálculo de taxas totais com a "apresentação", a aparência visual da fatura. Para avaliar esse defeito, imagine as alterações que seriam necessárias para elaborar a fatura em HTML para apresentação na Web. Reimplemente o programa usando uma classe `InvoiceFormatter` separada para formatar a fatura. Isto é, os métodos `Invoice` e `LineItem` não são mais responsáveis pela formatação. Entretanto, eles adquirirão outras responsabilidades, porque a classe `InvoiceFormatter` precisa consultá-los para obter os valores de que necessita.

★★★ **Exercício P12.4.** Escreva um programa que ensina aritmética ao seu irmão mais novo. O programa testa a adição e a subtração. No nível 1, ele testa apenas a adição de números menores que 10 cuja soma também é menor que 10. No nível 2, testa a adição de números arbitrários de um único dígito. No nível 3, ele testa a subtração de números de um único dígito cuja diferença seja não-negativa. Gere problemas aleatórios e obtenha a entrada do jogador. O jogador faz duas tentativas por problema. Avance de um nível para o seguinte quando o jogador alcançar cinco pontos.

★★★ **Exercício P12.5.** Projete um sistema simples de mensagens de e-mails. Uma mensagem tem um destinatário, um remetente e um texto de mensagem. Uma caixa postal pode armazenar mensagens. Forneça algumas caixas postais para diferentes usuários e uma interface para o usuário efetuar login, enviar mensagens para outros usuários, ler suas próprias mensagens e efetuar logout. Siga o processo de projeto descrito neste capítulo.

★★ **Exercício P12.6.** Escreva um programa que simula uma máquina de venda automática. Os produtos podem ser comprados inserindo moedas com um valor pelo menos igual ao do custo do produto. Um usuário seleciona um produto a partir de uma lista de produtos disponíveis, adiciona moedas e obtém o produto ou as moedas de volta caso tenha sido fornecido dinheiro insuficiente ou se não houver mais o produto. A máquina não fornece troco se for adicionado dinheiro a mais. O estoque de produtos pode ser renovado e o dinheiro removido por um operador. Siga o processo de projeto descrito neste capítulo. Sua solução deve incluir uma classe `VendingMachine` que não esteja associada com as classes `Scanner` ou `PrintStream`.

★★★ **Exercício P12.7.** Escreva um programa para projetar uma agenda de compromissos. Um compromisso inclui a data, o horário de início, o horário de término e uma descrição; por exemplo,

```
Dentist 2007/10/1 17:30 18:30
CS1 class 2007/10/2 08:30 10:00
```

Forneça uma interface para adicionar compromissos, remover compromissos cancelados e imprimir uma lista de compromissos para determinado dia. Siga o processo de projeto descrito neste capítulo. Sua solução deve incluir uma classe `AppointmentCalendar` que não esteja associada com as classes `Scanner` ou `PrintStream`.

★★★ **Exercício P12.8.** *Reserva de passagens aéreas.* Escreva um programa que faz reserva de passagens aéreas. Suponha que o avião tenha 20 poltronas na primeira classe (5 fileiras de 4 poltronas cada, separadas por um corredor) e 90 poltronas na classe econômica (15 fileiras de 6 poltronas cada, separadas por um corredor). Seu programa deve ter três comandos: adicionar passageiros, mostrar a poltrona e fechar. Quando os passageiros são adicionados, pergunte a classe (primeira classe ou econômica), o número de passageiros juntos (1 ou 2 na primeira classe; 1 a 3 na econômica) e a preferência em relação à poltrona (corredor ou janela na primeira classe; corredor, meio ou janela na econômica). Então tente localizar uma correspondência e determinar as poltronas. Se não existir correspondência, imprima uma mensagem. Sua solução deve incluir uma classe `Airplane` que não esteja associada com as classes `Scanner` ou `PrintSream`. Siga o processo de projeto descrito neste capítulo.

★★ **Exercício P12.9.** Modifique as implementações das classes no exemplo do ATM para que o banco gerencie uma coleção de contas bancárias e uma coleção de clientes separada. Permita contas-conjuntas em que algumas contas possam ter mais de um cliente.

★★★ **Exercício P12.10.** Escreva um programa que administra e avalia questionários. Um questionário consiste em perguntas. Há quatro tipos de perguntas: dissertativas, numéricas, com uma única resposta e de múltipla escolha. Ao avaliar uma questão dissertativa, ignore os espaços inicial e final e o uso de letras maiúsculas e minúsculas. Ao avaliar uma questão numérica, aceite uma solução que seja mais ou menos semelhante à da resposta.

Um questionário é especificado em um arquivo de texto. Cada questão inicia com uma letra que indica o tipo de pergunta (T, N, S, M), seguida por uma linha contendo o texto da pergunta. A próxima linha de uma questão que não seja uma de escolha contém a res-

posta. As perguntas de múltipla escolha têm uma série de respostas que são terminadas por uma linha em branco. Cada escolha inicia com + (correta) ou - (incorreta). Eis um arquivo de exemplo:

```
T
Which Java keyword is used to define a subclass?
extends
S
What is the original name of the Java language?
- *7
- C--
+ Oak
- Gosling

M
Which of the following types are supertypes of Rectangle?
- PrintStream
+ Shape
+ RectangularShape
+ Object
- String

N
What is the square root of 2?
1.41421356
```

Seu programa deve ler um arquivo de questionário, pedir ao usuário que responda todas as perguntas e avaliar as respostas. Siga o processo de projeto descrito neste capítulo.

★★★G **Exercício P12.11.** Implemente um programa para ensinar sua irmã a ler o relógio. No jogo, apresente um relógio analógico, como o da Figura 12. Gere horas aleatórias e exiba o relógio. Aceite suposições do jogador. Recompense o jogador pelas suposições corretas. Depois de duas suposições incorretas, exiba a resposta certa e apresente um novo horário aleatório. Implemente vários níveis de jogo. No nível 1, mostre apenas horas inteiras. No nível 2, mostre quartos de hora. No nível 3, mostre múltiplos de cinco minutos e, no nível 4, mostre qualquer número de minutos. Depois que o jogador alcançar cinco suposições certas em um nível, avance para o próximo nível.

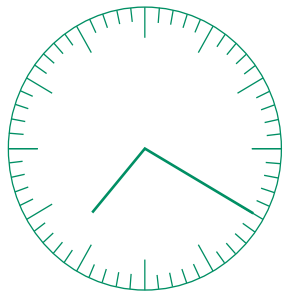


Figura 12 Um relógio analógico.

★★★G **Exercício P12.12.** Escreva um programa para projetar um cenário suburbano, com casas, ruas e carros. Os usuários podem adicionar casas e carros de várias cores a uma rua. Escreva requisitos mais específicos que incluam uma descrição detalhada da interface com o usuário. Em seguida, descubra classes e métodos, forneça diagramas UML e implemente seu programa.

★★★G **Exercício P12.13.** Escreva um editor gráfico simples que permite que os usuários adicionem uma mistura de formas (elipses, retângulos e linhas em cores diferentes) a um painel. Forneça comandos para carregar e salvar a figura. Descubra classes, forneça um diagrama UML e implemente seu programa.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 12.1.** Produza um documento de requisitos para um programa que permite que uma empresa envie mensagens personalizadas, por correio eletrônico ou serviço postal. Os arquivos de formato contêm o texto da mensagem, junto com campos variáveis (como Caro [Cargo] [Sobrenome]. . .). Um banco de dados (armazenado como um arquivo de texto) contém os valores de campo para cada destinatário. Utilize HTML como o formato de arquivo de saída. Então projete e implemente o programa.

★★★ **Projeto 12.2.** Escreva um jogo da velha que permite que um usuário humano jogue contra o computador. Seu programa jogará muitas partidas contra um oponente humano e aprenderá. Quando for a vez do computador jogar, ele selecionará aleatoriamente um campo vazio, mas nunca escolherá uma combinação perdedora. Para esse fim, seu programa deve manter um array de combinações de perdas. Sempre que o jogador humano ganhar, a combinação imediatamente anterior é armazenada como derrota. Por exemplo, suponha que `X` = computador e `0` = humano. Suponha que a combinação atual seja

0	X	X
	0	

Agora é a vez do jogador humano que, naturalmente, escolherá

0	X	X
	0	
		0

O computador então deve se lembrar da combinação anterior

0	X	X
	0	

como uma combinação perdedora. Como resultado, o computador nunca mais escolherá essa combinação a partir de

0	X	
	0	

ou

0		X
	0	

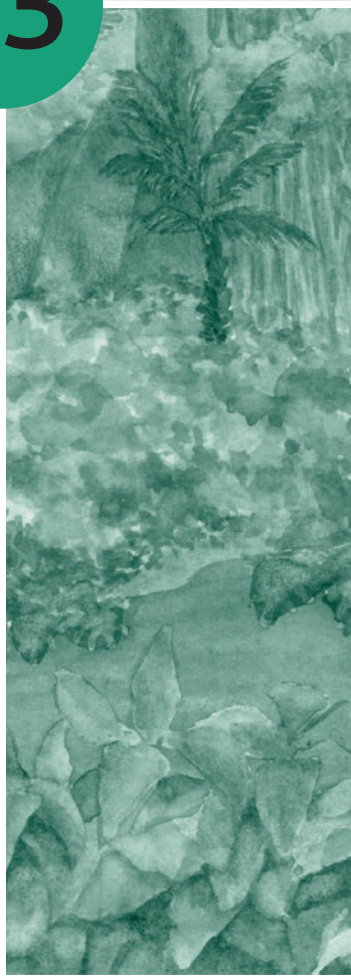
Descubra classes e forneça um diagrama UML antes de começar a programar.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. É pouco provável que o cliente faça um trabalho perfeito com o documento de requisitos. Se você não acomodar as mudanças, seu cliente pode não gostar do resultado. Se cobrar pelas alterações, seu cliente pode não gostar do custo.
2. Um modelo espiral "extremo", com uma grande quantidade de iterações.
3. Dar realimentação freqüente quer a iteração atual do produto seja ou não compatível com as necessidades do cliente.
4. `FileWriter`
5. Produzir o endereço de entrega do cliente.
6. Reformule as responsabilidades de modo que estejam em um nível mais alto ou apresentem mais classes para tratar as responsabilidades.
7. Por meio de agregação. O banco gerencia objetos do tipo conta bancária.
8. Por meio de herança.
9. As classes `BankAccount`, `System` e `PrintStream`.
10. A classe `Invoice` é responsável pelo cálculo do valor devido. Ela colabora com a classe `LineItem`.
11. Essa decisão de projeto reduz o acoplamento. Ela permite reutilizar as classes quando quisermos exibir a fatura em uma caixa de diálogo ou página Web.
12. O banco precisa armazenar a lista de clientes para que eles possam efetuar login. Precisamos localizar todas as contas bancárias de um cliente e escolhemos simplesmente armazená-las na classe de clientes. Nesse programa, não há mais a necessidade de acessar contas bancárias.
13. A classe `Bank` precisa ter uma responsabilidade adicional: carregar e salvar as contas. O banco pode assumir essa responsabilidade porque tem acesso aos objetos cliente e, por meio deles, às contas bancárias.

Capítulo 13

Recursão



Objetivos do capítulo

- Aprender sobre o método de recursão
- Entender o relacionamento entre recursão e iteração
- Analisar problemas que são muito mais fáceis de resolver por recursão do que por iteração
- Aprender a "pensar recursivamente"
- Saber utilizar métodos auxiliares recursivos
- Entender quando o uso de recursão afeta a eficiência de um algoritmo

O método de recursão é uma técnica poderosa para transformar problemas computacionais complexos em problemas mais simples. O termo "recursão" se refere ao fato de que a mesma computação recorre, ou ocorre repetidamente, à medida que o problema vai sendo resolvido. Em geral, a recursão é a maneira mais natural de se pensar em um problema e há alguns cálculos que são muito difíceis de realizar sem ela. Este capítulo mostra exemplos simples e complexos de recursão e ensina como "pensar recursivamente".

Conteúdo do Capítulo

13.1 Números triangulares 534
Erro comum 13.1: Recursão infinita 538
13.2 Permutações 538
Erro comum 13.2: Rastreando métodos recursivos 542
Como fazer 13.1: Pensando recursivamente 543
13.3 Métodos auxiliares recursivos 546
13.4 A eficiência da recursão 548
Fato aleatório 13.1: Os limites da computação

13.5 Recursões mútuas 554

13.1 Números triangulares

Começaremos este capítulo com um exemplo muito simples que mostrará o poder de pensar recursivamente. Neste exemplo, examinaremos formas triangulares como as da Seção 6.3. Gostaríamos de calcular a área de um triângulo de largura *n*, assumindo que cada quadrado `[]` tem área 1. Esse valor às vezes é chamado *n-ésimo número triangular.* Por exemplo, como você pode determinar ao examinar

```
[]
[][]
[][][]
```

o terceiro número triangular é 6.

Talvez você já saiba que há uma fórmula muito simples para calcular esses números, mas por enquanto deve fingir que não a conhece. O propósito final desta seção não é calcular números triangulares, mas aprender sobre o conceito de recursão em uma situação simples.

Eis o esboço da classe que desenvolveremos:

```
public class Triangle
{
   public Triangle(int aWidth)
   {
      width = aWidth;
   }
   public int getArea()
   {
      . . .
   }
   private int width;
}
```

Se a largura do triângulo for 1, então o triângulo consistirá em um único quadrado e sua área será 1. Vamos tratar desse caso primeiro.

```
public int getArea()
{
   if (width == 1) return 1;
   . . .
}
```

Para lidar com o caso geral, considere esta figura.

```
[]
[][]
[][][]
[][][][]
```

Suponha que soubéssemos a área do menor triângulo colorido. Então poderíamos facilmente calcular a área do triângulo maior como

```
smallerArea + width
```

Como podemos obter a área menor? Vamos fazer um triângulo menor e perguntar isso!

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Agora podemos completar o método `getArea`:

```
public int getArea()
{
   if (width == 1) return 1;
   Triangle smallerTriangle = new Triangle(width - 1);
   int smallerArea = smallerTriangle.getArea();
   return smallerArea + width;
}
```

Um cálculo recursivo resolve um problema utilizando a solução do mesmo problema com valores mais simples.

Eis uma ilustração do que acontece quando calculamos a área de um triângulo de largura 4.

- O método `getArea` faz um triângulo menor, de largura 3.
  - Ele chama `getArea` nesse triângulo.
    - Esse método faz um triângulo menor, de largura 2.
      - Ele chama `getArea` nesse triângulo.
        - Esse método faz um triângulo menor, de largura 1.
        - Ele chama `getArea` nesse triângulo.
      - Esse método retorna 1.
    - O método retorna `smallerArea + width` = 1 + 2 = 3.
  - O método retorna `smallerArea + width` = 3 + 3 = 6.
- O método retorna `smallerArea + width` = 6 + 4 = 10.

Essa solução tem um aspecto notável. Para resolver o problema de área de um triângulo de uma dada largura, utilizamos o fato de que podemos resolver o mesmo problema para uma largura menor. Isso é chamado solução *recursiva*.

O padrão de chamada de um *método recursivo* parece complicado, e o segredo para o projeto bem-sucedido de um método recursivo é *não pensar no assunto*. Em vez disso, examine o método `getArea` mais uma vez e note como ele é bem razoável. Se a largura for 1, então, é claro que a área será 1. A próxima parte é igualmente razoável. Calcule a área do triângulo menor *e não pense por que isso funciona*. Então a área do triângulo maior é claramente a soma da área menor e da largura.

Há dois requisitos-chave para nos certificarmos de que a recursão é bem-sucedida:

- Toda chamada recursiva deve simplificar o cálculo de alguma maneira.
- Há casos especiais para tratar as computações mais simples diretamente.

Para que uma recursão termine, deve haver casos especiais para os valores mais simples.

O método `getArea` chama a si próprio novamente com valores de largura cada vez menores. Por fim, a largura deve alcançar 1 e há um caso especial para o cálculo da área de um triângulo com largura 1. Portanto, o método `getArea` é sempre bem-sucedido.

Na verdade, você precisa tomar cuidado. O que acontece ao chamar a área de um triângulo com largura –1? Ele calcula a área de um triângulo com largura –2, que calcula a área de um triângulo com largura –3 e assim por diante. Para evitar isso, o método `getArea` deve retornar 0 se a largura for ≤ 0.

A recursão não é realmente necessária para calcular os números triangulares. A área de um triângulo é igual à soma:

```
1 + 2 + 3 + . . . + width
```

Naturalmente, você pode programar um laço simples:

```
double area = 0;
for (int i = 1; i <= width; i++)
   area = area + i;
```

Muitas recursões simples podem ser calculadas como laços. Entretanto, laços equivalentes para recursões mais complexas – como o do nosso próximo exemplo – podem ser complexos.

Na verdade, nesse caso, você nem precisa de um laço para calcular a resposta. A soma dos primeiros n inteiros pode ser calculada como

$$1 + 2 + \cdots + n = n \times (n + 1) / 2$$

Portanto, a área é igual a

```
width * (width + 1) / 2
```

Assim, nem a recursão nem um laço são necessários para resolver esse problema. A solução recursiva foi projetada apenas como um "aquecimento" para introduzi-lo ao conceito de recursão.

ch13/triangle/Triangle.java

```
1  /**
2     Uma forma triangular composta de quadrados por unidade empilhados como esta:
3     []
4     [][]
5     [][][]
6     . . .
7  */
8  public class Triangle
9  {
```

```
/**
      Constrói uma forma triangular.
      @param aWidth largura (e altura) do triângulo
   */
   public Triangle(int aWidth)
   {
      width = aWidth;
   }

   /**
      Calcula a área do triângulo.
      @return área
   */
   public int getArea()
   {
      if (width <= 0) return 0;
      if (width == 1) return 1;
      Triangle smallerTriangle = new Triangle(width - 1);
      int smallerArea = smallerTriangle.getArea();
      return smallerArea + width;
   }

   private int width;
}
```

ch13/triangle/TriangleTester.java

```
public class TriangleTester
{
   public static void main(String[] args)
   {
      Triangle t = new Triangle(10);
      int area = t.getArea();
      System.out.println("Area: " + area);
      System.out.println("Expected: 55");
   }
}
```

Saída

```
Enter width: 10
Area: 55
Expected: 55
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Por que a instrução `if (width == 1) return 1;` no método `getArea` é desnecessária?
2. Como você modificaria o programa para calcular recursivamente a área de um quadrado?

Erro comum 13.1

Recursão infinita

Um erro de programação comum é uma recursão infinita: um método que chama a si próprio repetidamente e sem condição de término. O computador precisa de uma quantidade de memória para contabilizar cada chamada a um método. Depois de um número de chamadas, toda memória disponível para esse propósito é consumida. Seu programa trava e informa uma "falha de pilha" (*stack fault*).

A recursão infinita acontece porque os valores de parâmetro não se tornam mais simples ou porque está faltando um caso de término especial. Por exemplo, suponha que o método `getArea` calcule a área de um triângulo com largura 0. Se não fosse pelo teste especial, o método teria construído triângulos com largura –1, –2, –3 e assim por diante.

13.2 Permutações

Agora voltaremos nossa atenção para um exemplo de recursão mais complexo que seria difícil de programar com um laço simples. Projetaremos uma classe que lista todas as permutações de uma string. Permutação é simplesmente um rearranjo das letras. Por exemplo, a string `"eat"` tem seis permutações (inclusive a própria string original):

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

Como na seção anterior, definiremos uma classe que é responsável pelo cálculo da resposta. Nesse caso, a resposta não é um único número, mas uma coleção de strings permutadas. Eis nossa classe:

```
public class PermutationGenerator
{
   public PermutationGenerator(String aWord) { . . . }
   ArrayList<String> getPermutations() { . . . }
}
```

Eis o programa de teste que imprime todas as permutações da string `"eat"`:

ch13/permute/PermutationGeneratorDemo.java

```
1  import java.util.ArrayList;
2
3  /**
4     Esse programa demonstra o gerador de permutações.
5  */
6  public class PermutationGeneratorDemo
7  {
```

```
   public static void main(String[] args)
   {
      PermutationGenerator generator
            = new PermutationGenerator("eat");
      ArrayList<String> permutations = generator.getPermutations();
      for (String s : permutations)
      {
         System.out.println(s);
      }
   }
}
```

Saída

```
eat
eta
aet
ate
tea
tae
```

Agora precisamos de uma maneira de gerar as permutações recursivamente. Considere a string `"eat"`. Vamos simplificar o problema. Primeiro, geraremos todas as permutações que iniciam com a letra `'e'`, depois aquelas que iniciam com `'a'` e, por fim, aquelas que iniciam com `'t'`. Como geramos as permutações que iniciam com `'e'`? Precisamos conhecer as permutações da substring `"at"`. Mas esse é o mesmo problema – gerar todas as permutações – com uma entrada mais simples, a saber, a string menor `"at"`. Portanto, você pode utilizar recursão. Gere as permutações da substring `"at"`. Elas são:

```
"at"
"ta"
```

Para cada permutação dessa substring, coloque na frente a letra `'e'` para obter as permutações de `"eat"` que iniciam com `'e'`, a saber:

```
"eat"
"eta"
```

Agora voltemos para as permutações de `"eat"` que iniciam com `'a'`. Precisamos produzir as permutações das letras restantes, `"et"`. Elas são

```
"et"
"te"
```

Adicionamos a letra `'a'` à frente das strings e obtemos

```
"aet"
"ate"
```

Geramos as permutações que iniciam com `'t'` da mesma maneira.

Essa é a idéia. A implementação é relativamente simples e direta. No método `getPermutations`, fazemos um laço que percorre por todas as posições na palavra a ser permutada. Para cada uma delas, calculamos a palavra mais curta que é obtida removendo a `i`-ésima letra:

```
String shorterWord = word.substring(0, i) + word.substring(i + 1);
```

Construímos um gerador de permutações para obter as permutações da palavra mais curta e pedimos que ele nos forneça todas as permutações desta palavra mais curta.

```
PermutationGenerator shorterPermutationGenerator
      = new PermutationGenerator(shorterWord);
ArrayList<String> shorterWordPermutations
      = shorterPermutationGenerator.getPermutations();
```

Por fim, adicionamos a letra removida na frente de todas as permutações da palavra mais curta.

```
for (String s : shorterWordPermutations)
{
   result.add(word.charAt(i) + s);
}
```

Como sempre, precisamos fornecer um caso especial para as strings mais simples. A possível string mais simples é a string vazia, que tem uma única permutação – ela própria.

Eis a classe `PermutationGenerator` completa.

ch13/permute/PermutationGenerator.java

```
1  import java.util.ArrayList;
2
3  /**
4     Essa classe gera permutações de uma palavra.
5  */
6  public class PermutationGenerator
7  {
8     /**
9        Constrói um gerador de permutações.
10       @param aWord palavra a permutar
11    */
12    public PermutationGenerator(String aWord)
13    {
14       word = aWord;
15    }
16
17    /**
18       Obtém todas as permutações de uma dada palavra.
19    */
20    public ArrayList<String> getPermutations()
21    {
22       ArrayList<String> result = new ArrayList<String>();
23
24       // A string vazia tem uma única permutação: ela própria
25       if (word.length() == 0)
26       {
27          result.add(word);
28          return result;
29       }
```

```
30
31        // Itera por todos os caracteres da palavra
32        for (int i = 0; i < word.length(); i++)
33        {
34           // Forma uma palavra mais simples removendo o i-ésimo caractere
35           String shorterWord = word.substring(0, i)
36                 + word.substring(i + 1);
37
38           // Gera todas as permutações da palavra mais simples
39           PermutationGenerator shorterPermutationGenerator
40                 = new PermutationGenerator(shorterWord);
41           ArrayList<String> shorterWordPermutations
42                 = shorterPermutationGenerator.getPermutations();
43
44           // Adiciona o caractere removido à frente de
45           // cada permutação da palavra mais simples
46           for (String s : shorterWordPermutations)
47           {
48              result.add(word.charAt(i) + s);
49           }
50        }
51        // Retorna todas as permutações
52        return result;
53     }
54
55     private String word;
56  }
```

Compare as classes `PermutationGenerator` e `Triangle`. Ambas têm o mesmo princípio. Quando elas têm uma entrada mais complexa, primeiro resolvem o problema de uma entrada mais simples. Então combinam o resultado da entrada mais simples com trabalho adicional para apresentar os resultados para a entrada mais complexa. Não há realmente uma complexidade particular por atrás desse processo, contanto que você pense na solução apenas nesse nível. Entretanto, nos bastidores, a entrada mais simples cria entradas ainda mais simples, o que cria ainda outra simplificação e assim por diante, até que uma entrada seja tão simples que o resultado possa ser obtido sem mais nenhuma ajuda. É interessante pensar nesse processo, mas isso também pode ser confuso. O importante é que é possível se concentrar só no nível que interessa – formando uma solução a partir do problema ligeiramente mais simples, ignorando o fato de que ele também usa recursão para obter seus resultados.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Quais são todas as permutações da palavra de quatro letras `beat`?
4. Nossa recursão para o gerador de permutações pára na string vazia. Que modificação simples faria a recursão parar nas strings de comprimento 0 ou 1?

ERRO COMUM 13.2

Rastreando métodos recursivos

Depurar um método recursivo pode ser um desafio. Quando você configura um ponto de interrupção em um método recursivo, o programa pára logo que essa linha de programa é encontrada em *qualquer chamada do método recursivo*. Suponha que você queira depurar o método recursivo `getArea` da classe `Triangle`. Depure o programa `TriangleTester` e execute até o começo do método `getArea`. Inspecione a variável de instância `width`. O valor do método é 10.

Remova o ponto de interrupção e agora execute até a instrução `return smallerArea + width;` (veja Figura 1). Quando você inspecionar `width` novamente, seu valor será 2! Isso não faz sentido. Não houve uma instrução que alterasse o valor de `width`. Esse é um bug do depurador?

Não. O programa parou na primeira chamada recursiva a `getArea` que alcançou a instrução `return`. Se estiver confuso, examine a *pilha de chamadas* (parte superior esquerda da figura). Você verá que nove chamadas para `getArea` estão pendentes.

Você pode depurar métodos recursivos usando o depurador. Você só precisa ser muito cuidadoso e observar a pilha de chamadas para entender em que chamada aninhada você está atualmente.

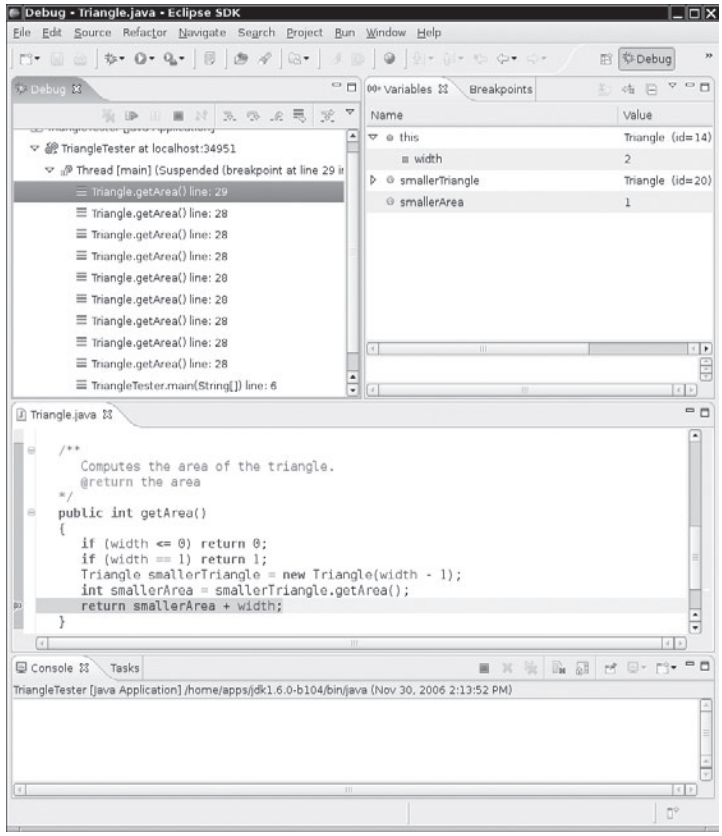


Figura 1 Depurando um método recursivo.

COMO FAZER 13.1

Pensando recursivamente

Resolver um problema recursivamente exige uma disposição mental diferente daquela necessária para resolver por meio de laços de programação. De fato, ajuda se você for, ou fingir ser, um pouco preguiçoso e gostar que outros façam a maior parte do trabalho para você. Se precisar resolver um problema complexo, finja que "outra pessoa" fará a maior parte do trabalho pesado e resolva o problema de todas as entradas mais simples. Então você só precisa descobrir como transformar soluções com entradas mais simples em uma solução para todo o problema.

Para ilustrar o método de recursão, vamos considerar o seguinte problema. Queremos testar se uma frase é um *palíndromo* – uma string que é igual a si mesma quando você inverte todos os caracteres. Exemplos típicos de palíndromos são

- A man, a plan, a canal – Panama!
- Go hang a salami, I'm a lasagna hog

e, naturalmente, o palíndromo mais antigo de todos:

- Madam, I'm Adam

Ao verificar um palíndromo, correspondemos as letras maiúsculas e minúsculas e ignoramos todos os espaços e marcas de pontuação.

Precisamos implementar o método `isPalindrome` na seguinte classe:

```
public class Sentence
{
   /**
      Constrói uma frase.
      @param aText string contendo todos os caracteres da frase
   */
   public Sentence(String aText)
   {
       text = aText;
   }
   /**
      Testa se essa frase é um palíndromo.
      @return true se essa frase for um palíndromo, false caso contrário
   */
   public boolean isPalindrome()
   {
      . . .
   }
   private String text;
}
```

Passo 1 Considere várias maneiras de simplificar entradas.

Corrija mentalmente uma entrada particular ou conjunto de entradas do problema que você quer resolver.

Pense em como você pode simplificar as entradas de maneira que o mesmo problema possa ser aplicado à entrada mais simples.

Ao considerar entradas mais simples, você pode querer remover somente uma pequena parte da entrada original – talvez remover um ou dois caracteres de uma string ou uma parte pequena de uma forma geométrica. Mas às vezes é mais útil cortar a entrada pela metade e então ver o que significa resolver o problema para ambas as metades.

No problema de teste de palíndromo, a entrada é a string que precisamos testar. Como simplificar a entrada? Eis várias possibilidades:

- Remover o primeiro caractere.
- Remover o último caractere.
- Remover tanto o primeiro como o último caractere.
- Remover um caractere do meio.
- Dividir a string ao meio.

Essas entradas mais simples são potenciais entradas para o teste de palíndromo.

Passo 2 Combine soluções com entradas mais simples em uma solução do problema original.

Considere mentalmente as soluções de seu problema para as entradas mais simples descobertas no Passo 1. Não se preocupe *como* essas soluções serão obtidas. Simplesmente acredite que as soluções estão prontamente disponíveis. Apenas diga a você mesmo: essas são entradas mais simples, então outra pessoa resolverá o problema para mim.

Agora pense como você pode tornar a solução das entradas mais simples em uma solução para a entrada que você está considerando atualmente. Talvez seja necessário adicionar uma pequena quantidade, relacionada àquela que você removeu para chegar à entrada mais simples. Talvez você divida a entrada original ao meio e tenha soluções para cada metade. Então você pode precisar somar ambas as soluções para chegar a uma solução para o todo.

Considere os métodos para simplificar as entradas do teste de palíndromo. Dividir a string ao meio não parece uma boa idéia. Se dividir

```
"Madam, I'm Adam"
```

ao meio, você obtém duas strings:

```
"Madam, I"
```

e

```
"'m Adam"
```

Nenhuma delas é um palíndromo. Dividir a entrada ao meio e testar se as metades são palíndromos parece um beco sem saída.

A simplificação mais promissora é remover o primeiro *e* o último caractere.

Remover o primeiro `M` e o último `m` produz

```
"adam, I'm Ada"
```

Suponha que você possa verificar se a string mais curta é um palíndromo. Então *naturalmente* a string original é um palíndromo – colocamos a mesma letra na frente e atrás. Isso é extremamente promissor. Uma palavra é um palíndromo se

- A primeira e a última letra são iguais (ignorando o uso de maiúsculas e minúsculas)

e

- A palavra obtida removendo a primeira e a última letra é um palíndromo.

Novamente, não se preocupe como o teste funciona para a string mais curta. Ele simplesmente funciona.

Há um outro caso a ser considerado. E se a primeira ou a última letra da palavra não for uma letra? Por exemplo, a string

```
"A man, a plan, a canal, Panama!"
```

termina com um caractere `!`, que não corresponde ao `A` inicial. Mas devemos ignorar caracteres que não são letras quando fazemos testes para localizar palíndromos. Portanto, quando o último caractere não for uma letra, mas o primeiro for, não faz sentido remover o primeiro e o último caracteres. Isso não é um problema. Remova somente o último caractere. Se a string mais curta for um palíndromo, então ela será um palíndromo quando você incluir um caracter que não for letra.

O mesmo argumento se aplica se o primeiro caractere não for uma letra. Agora temos um conjunto completo de casos.

- Se o primeiro e o último caractere forem letras, então verifique se eles se correspondem. Se sim, remova os dois e teste a string mais curta.
- Caso contrário, se o último caractere não for uma letra, remova-o e teste a string mais curta.
- Caso contrário, o primeiro caractere não é uma letra. Remova-o e teste a string mais curta.

Nos três casos, você pode utilizar a solução do problema mais simples para chegar a uma solução para o seu problema.

Passo 3 Descubra soluções para as entradas mais simples.

Uma computação recursiva continua simplificando suas entradas. Por fim, ela chega a entradas muito simples. Para certificar-se de que a recursão chega ao fim, você deve lidar com as entradas simples separadamente. Apresente soluções especiais para elas, o que normalmente é muito fácil.

Entretanto, às vezes você se envolve em perguntas filosóficas ao lidar com entradas *degeneradas*: Strings vazias, formas sem área e assim por diante. Então você pode querer investigar uma entrada ligeiramente maior que é reduzida a uma entrada trivial e ver o valor que você deve anexar às entradas degeneradas de modo que o valor mais simples, quando utilizado de acordo com as regras que você descobriu no Passo 2, forneça a resposta correta.

Vejamos as strings mais simples para o teste de palíndromo:

- Strings com dois caracteres
- Strings com um único caractere
- A string vazia

Não precisamos propor uma solução especial para strings com dois caracteres. O Passo 2 ainda se aplica a essas strings – qualquer um dos caracteres, ou ambos, são removidos. Mas precisamos nos preocupar com as strings de comprimento 0 e 1. Nesses casos, não é possível aplicar o Passo 2. Não há dois caracteres para remover.

A string vazia é um palíndromo – é a mesma string quando você a lê de trás para frente. Se achar isso muito artificial, considere uma string `"mm"`. De acordo com a regra descoberta no Passo 2, essa string é um palíndromo se o primeiro e último caracteres dessa string corresponderem, e o restante – isto é, a string vazia – também for um palíndromo. Portanto, faz sentido considerar a string vazia um palíndromo.

Uma string com uma única letra, como `"I"`, é um palíndromo. E o que dizer do caso em que o caractere não é uma letra, como`"!"`? Remover o `!` cria a string vazia, que é um palíndromo. Portanto, concluímos que todas as strings de comprimento 0 ou 1 são palíndromos.

Passo 4 Implemente a solução combinando os casos simples e o passo de redução.

Agora você está pronto para implementar a solução. Crie casos separados para as entradas simples que você considerou no Passo 3. Se a entrada não for um dos casos mais simples, então implementa a lógica que você descobriu no Passo 2.

Eis o método `isPalindrome`.

```
public boolean isPalindrome()
{
   int length = text.length();
   // Caso separado das strings mais curtas.
   if (length <= 1) return true;
   // Converte o primeiro e o último caractere em letras minúsculas.
   char first = Character.toLowerCase(text.charAt(0));
   char last = Character.toLowerCase(text.charAt(length - 1));
   if (Character.isLetter(first) && Character.isLetter(last))
   {
      // Ambos são letras.
      if (first == last)
      {
         // Remove tanto o primeiro como o último caractere.
         Sentence shorter = new Sentence(text.substring(1, length - 1));
         return shorter.isPalindrome();
      }
      else
         return false;
   }
   else if (!Character.isLetter(last))
   {
      // Remove o último caractere.
      Sentence shorter = new Sentence(text.substring(0, length - 1));
      return shorter.isPalindrome();
   }
   else
   {
      // Remove o primeiro caractere.
      Sentence shorter = new Sentence(text.substring(1));
      return shorter.isPalindrome();
   }
}
```

13.3 Métodos auxiliares recursivos

Às vezes é mais fácil encontrar uma solução recursiva se você fizer uma alteração sutil no problema original.

Às vezes é mais fácil encontrar uma solução recursiva se você alterar sutilmente o problema original. Então o problema original pode ser resolvido chamando um método auxiliar recursivo.

Eis um exemplo típico. Considere o teste de palíndromo da seção Como Fazer 13.1. É um pouco ineficiente construir novos objetos `Sentence` em cada passo. Agora considere a seguinte alteração no problema. Em vez de testar se a frase inteira é um palíndromo, vamos verificar se uma substring é um palíndromo:

```
/**
   Testa se uma substring da frase é um palíndromo.
   @param start índice do primeiro caractere da substring
   @param end índice do último caractere da substring
   @return true se a substring for um palíndromo
*/
public boolean isPalindrome(int start, int end)
```

Esse método mostrou-se ainda mais fácil de implementar do que o teste original. Nas chamadas recursivas, simplesmente ajuste os parâmetros `start` e `end` para pular pares de letras correspondentes e caracteres que não são letras. Não há não necessidade de construir novos objetos `Sentence` para representar as strings mais curtas.

```
public boolean isPalindrome(int start, int end)
{
   // Caso separado para substrings de comprimento 0 e 1.
   if (start >= end) return true;

   // Converte o primeiro e o último caractere em letras minúsculas.
   char first = Character.toLowerCase(text.charAt(start));
   char last = Character.toLowerCase(text.charAt(end));

   if (Character.isLetter(first) && Character.isLetter(last))
   {
      if (first == last)
      {
         // Testa substring que não contém letras correspondentes.
         return isPalindrome(start + 1, end - 1);
      }
      else
         return false;
   }
   else if (!Character.isLetter(last))
   {
      // Testa substring que não contém o último caractere.
      return isPalindrome(start, end - 1);
   }
   else
   {
      // Testa substring que não contém o primeiro caractere.
      return isPalindrome(start + 1, end);
   }
}
```

Você ainda deve fornecer um método para resolver todo o problema – o usuário de seu método não precisa conhecer o truque com as posições de substring. Simplesmente chame o método auxiliar com posições que testam a string inteira:

```
public boolean isPalindrome()
{
   return isPalindrome(0, text.length() - 1);
}
```

Observe que essa chamada *não* é um método recursivo. O método `isPalindrome()` chama o método auxiliar `isPalindrome(int, int)`. Nesse exemplo, utilizamos sobrecarga para definir dois métodos com o mesmo nome. O método `isPalindrome` sem parâmetros é o

método que esperamos que o público utilize. O segundo método, com dois parâmetros `int`, é o método auxiliar recursivo. Se preferir, você pode evitar métodos sobrecarregados escolhendo um nome diferente para o método auxiliar, como `substringIsPalindrome`.

Utilize a técnica de métodos auxiliares recursivos sempre que for mais fácil resolver um problema recursivo que é um pouco diferente do problema original.

Autoverificação da aprendizagem

5. Precisamos dar o mesmo nome para os dois métodos `isPalindrome`?
6. Quando o método `isPalindrome` recursivo pára de chamar a si próprio?

13.4 A eficiência da recursão

Como você viu neste capítulo, a recursão pode ser uma ferramenta poderosa para implementar algoritmos complexos. Por outro lado, a recursão pode levar a algoritmos que apresentam um desempenho ruim. Nesta seção, analisaremos a pergunta de quando a recursão é benéfica e quando é ineficiente.

Considere a seqüência de Fibonacci introduzida no Exercício P6.4, uma seqüência de números definida pela equação

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Isto é, cada valor da seqüência é a soma dos dois valores precedentes. Os dez primeiros termos da seqüência são

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

É fácil estender essa seqüência indefinidamente. Basta continuar acrescentando a soma dos dois últimos valores da seqüência. Por exemplo, a próxima entrada é 34 + 55 = 89.

Gostaríamos de escrever uma função que calculasse f_n para qualquer valor de n. Vamos converter a definição diretamente em um método recursivo:

ch13/fib/RecursiveFib.java

```
import java.util.Scanner;

/**
    Esse programa calcula números de Fibonacci utilizando um
    método recursivo.
*/
```

```
public class RecursiveFib
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++)
      {
         long f = fib(i);
         System.out.println("fib(" + i + ") = " + f);
      }
   }

   /**
      Calcula um número de Fibonacci.
      @param n um inteiro
      @return n-ésimo número de Fibonacci
   */
   public static long fib(int n)
   {
      if (n <= 2) return 1;
      else return fib(n - 1) + fib(n - 2);
   }
}
```

Saída

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Certamente isso é simples e o método funcionará corretamente. Mas note a saída atentamente enquanto você executa o programa de teste. As primeiras poucas chamadas para o método `fib` são bem rápidas. Mas, para valores maiores, o programa demora um tempo espantosamente longo entre as saídas.

Isso não faz sentido. Munido de lápis, papel e calculadora de bolso você poderia calcular esses números com bastante rapidez, então isso não poderia exigir tanto do computador.

Para descobrir o problema, vamos inserir mensagens de rastreamento no método:

ch13/fib/RecursiveFibTracer.java

```
1  import java.util.Scanner;
2
3  /**
4     Esse programa imprime mensagens de rastreamento que mostram a freqüência com que
5     o método recursivo chama a si próprio para calcular números de Fibonacci.
6  */
7  public class RecursiveFibTracer
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12       System.out.print("Enter n: ");
13       int n = in.nextInt();
14
15       long f = fib(n);
16
17       System.out.println("fib(" + n + ") = " + f);
18    }
19
20    /**
21       Calcula um número de Fibonacci.
22       @param n um inteiro
23       @return n-ésimo número de Fibonacci
24    */
25    public static long fib(int n)
26    {
27       System.out.println("Entering fib: n = " + n);
28       long f;
29       if (n <= 2) f = 1;
30       else f = fib(n - 1) + fib(n - 2);
31       System.out.println("Exiting fib: n = " + n
32             + " return value = " + f);
33       return f;
34    }
35 }
```

Saída

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
```

```
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

A Figura 2 mostra a árvore de chamada para calcular `fib(6)`. Agora está ficando evidente por que o método leva tanto tempo. Ele está calculando os mesmos valores repetidamente. Por exemplo, o cálculo de `fib(6)` chama `fib(4)` duas vezes e `fib(3)` três vezes. Isso é muito diferente do cálculo que nós faríamos com lápis e papel. Neste caso, simplesmente anotamos os valores à medida que são calculados e somamos os dois últimos para obter o próximo até alcançarmos a entrada desejada; jamais um valor da seqüência será calculado duas vezes.

Se imitarmos o processo baseado em lápis e papel, então obtemos o seguinte programa.

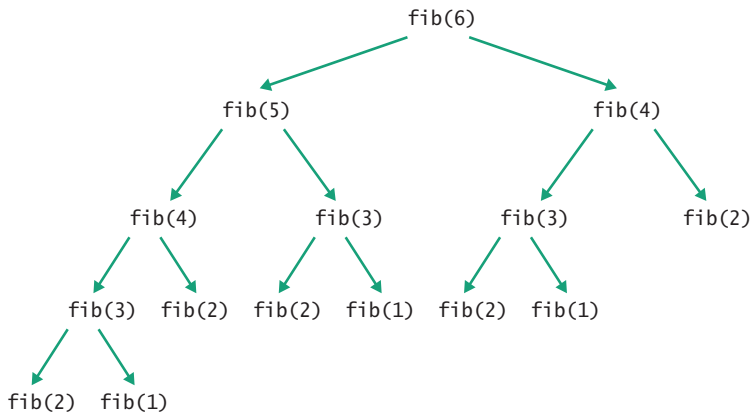


Figura 2 Padrão de chamada do método `fib` recursivo.

ch13/fib/LoopFib.java

```
1  import java.util.Scanner;
2
3  /**
4     Esse programa calcula números de Fibonacci utilizando um método iterativo.
5  */
6  public class LoopFib
7  {
8     public static void main(String[] args)
9     {
10       Scanner in = new Scanner(System.in);
11       System.out.print("Enter n: ");
12       int n = in.nextInt();
13
14       for (int i = 1; i <= n; i++)
15       {
16          long f = fib(i);
17          System.out.println("fib(" + i + ") = " + f);
18       }
19    }
20
21    /**
22       Calcula um número de Fibonacci.
23       @param n um inteiro
24       @return n-ésimo número de Fibonacci
25    */
26    public static long fib(int n)
27    {
28       if (n <= 2) return 1;
29       long fold = 1;
30       long fold2 = 1;
31       long fnew = 1;
32       for (int i = 3; i <= n; i++)
33       {
34          fnew = fold + fold2;
35          fold2 = fold;
36          fold = fnew;
37       }
38       return fnew;
39    }
40 }
```

Saída

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Esse método executa *muito* mais rápido que a versão recursiva.

Nesse exemplo do método `fib`, a solução recursiva foi fácil de programar porque seguiu exatamente a definição matemática, mas executou muito mais lentamente do que a solução iterativa, pois computou muitos resultados intermediários várias vezes.

É possível acelerar uma solução recursiva transformando-a em um laço? Em geral, a solução iterativa e a recursiva têm essencialmente o mesmo desempenho. Por exemplo, eis uma solução iterativa para o teste de palíndromo.

```
public boolean isPalindrome()
{
   int start = 0;
   int end = text.length() - 1;
   while (start < end)
   {
      char first = Character.toLowerCase(text.charAt(start));
      char last = Character.toLowerCase(text.charAt(end);

      if (Character.isLetter(first) && Character.isLetter(last))
      {
         // Ambos os caracteres são letras.
         if (first == last)
         {
            start++;
            end--;
         }
         else
            return false;
      }
      if (!Character.isLetter(last))
         end--;
      if (!Character.isLetter(first))
         start++;
   }
   return true;
}
```

Essa solução mantém duas variáveis de índice: `start` e `end`. O primeiro índice inicia no começo da string e é incrementado sempre que há uma correspondência de letra ou ignorado sempre que o caracter não for uma letra. O segundo índice inicia no fim da string e desloca-se para o início. Quando as duas variáveis de índice se encontram, a iteração pára.

Ocasionalmente, uma solução recursiva executa muito mais lentamente do que sua correspondente iterativa. Entretanto, na maioria dos casos, a solução recursiva só é um pouco mais lenta.

A iteração e a recursão executam quase na mesma velocidade. Se um palíndromo tiver n caracteres, a iteração executa o laço entre $n/2$ e n vezes, dependendo de quantos caracteres forem letras, visto que uma ou ambas as variáveis de índice são movidas em cada passo. De maneira semelhante, a solução recursiva chama a si própria entre $n/2$ e n vezes, porque um ou dois caracteres são removidos em cada passo.

Em tal situação, a solução iterativa tende a ser um pouco mais rápida, pois cada chamada de método recursivo demanda um certo tempo do processador. A princípio, é possível que um compilador esperto evite chamadas recursivas a métodos se seguir padrões simples, mas a maioria dos compiladores não faz isso. Desse ponto de vista, uma solução iterativa é preferível.

Há muitos problemas que são muito mais fáceis de resolver recursivamente do que iterativamente. Por exemplo, uma solução não-recursiva para o gerador de permutações não é absolutamente óbvia. Como o Exercício P13.11 mostra, é possível evitar a recursão, mas a solução resultante é bem complexa (e não é mais rápida).

Em muitos casos, uma solução recursiva é mais fácil de entender e implementar corretamente do que uma solução iterativa.

Em geral, soluções recursivas são mais fáceis de entender e implementar corretamente do que sua correspondente iterativa. Há certa elegância e economia em pensar em soluções recursivas, o que as tornam mais atrativas. Como disse o cientista da computação (e criador do interpretador GhostScript da linguagem de descrição gráfica PostScript) L. Peter Deutsch: "Iterar é humano, recorrer é divino."

Autoverificação da aprendizagem

7. Você pode calcular a função fatorial com um laço, utilizando a definição $n! = 1 \times 2 \times \ldots \times n$, ou recursivamente, utilizando a definição $0! = 1$ e $n! = (n-1)! \times n$. A abordagem recursiva é ineficiente nesse caso?
8. Por que não é fácil desenvolver uma solução iterativa para o gerador de permutações?

Fato aleatório 13.1

Os limites da computação

O Fato Aleatório 13.1 discute problemas que estão intrinsecamente além das capacidades de qualquer computador. Por exemplo, cientistas da computação teóricos provaram que é impossível escrever um programa que avalie seu dever de casa de programação comparando-o com a solução do instrutor e diga com certeza se esses dois programas sempre produzem os mesmos resultados para as mesmas entradas.

13.5 Recursões mútuas

Em uma recursão mútua, um conjunto de métodos cooperativos chama um ao outro repetidamente.

Nos exemplos precedentes, um método chamava a si próprio para resolver um problema mais simples. Às vezes, um conjunto de métodos cooperativos chama um ao outro de modo recursivo. Nesta seção, exploraremos uma típica situação dessa recursão mútua. Essa técnica é significativamente mais avançada do que a recursão simples discutida nas seções anteriores. Fique à vontade para pular esta seção se este for seu primeiro contato com recursão.

Desenvolveremos um programa que pode calcular os valores de expressões aritméticas como

```
3+4*5
(3+4)*5
1-(2-(3-(4-5)))
```

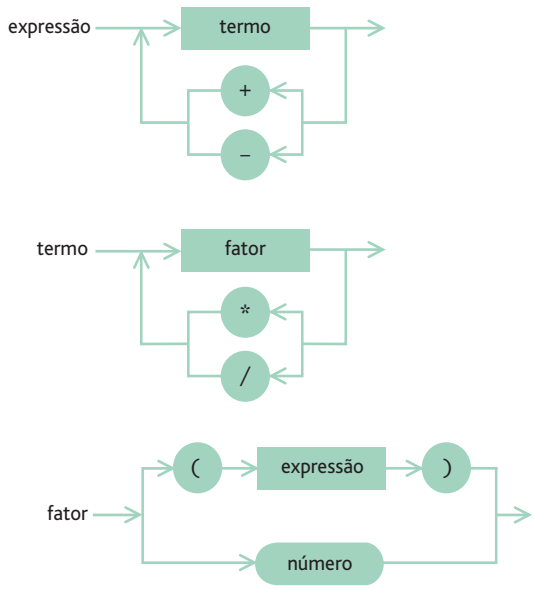


Figura 3 Diagramas de sintaxe para avaliar uma expressão.

Calcular tal expressão é complicado pelo fato de que `*` e `/` têm precedência sobre `+` e `-`, e que parênteses podem ser utilizados para agrupar subexpressões.

A Figura 3 mostra um conjunto de diagramas que descrevem a sintaxe dessas expressões. Para ver como os diagramas de sintaxe funcionam, considere a expressão `3+4*5`. Quando você entra no diagrama de sintaxe *expressão*, a seta aponta diretamente para *termo*, deixando-o sem alternativa, exceto entrar no diagrama de sintaxe *termo*. A seta aponta para *fator*, novamente deixando-o sem escolha. Depois de entrar no diagrama *fator*, você tem duas escolhas: seguir o desvio superior ou o desvio inferior. Como a primeira parte da expressão (ou token) de entrada é o número `3`, e não um `(`, você deve seguir o desvio inferior. Você aceita o token de entrada porque ele corresponde ao *número*. Siga a seta de *número* até o fim de *fator*. Assim como em uma chamada de método, agora você recua, retornando ao fim do elemento *fator* do diagrama *termo*. Agora você tem outra escolha – fazer um laço de volta no diagrama *termo* ou sair. O próximo token de entrada é um `+` e não corresponde nem ao `*` nem a `/` que seria necessário para fazer um laço de volta. Então você sai, retornando à *expressão*. Novamente, você tem uma escolha, fazer um laço de volta ou sair. Agora o `+` corresponde a uma das escolhas no laço. Aceite o `+` na entrada e volte para o elemento *termo*.

Dessa maneira, uma expressão é dividida em uma seqüência de termos, separados por `+` ou `-`, cada termo é dividido em uma seqüência de fatores, cada um é separado por `*` ou `/` e cada fator é uma expressão entre parênteses ou um número. Você pode representar essa divisão como uma árvore. A Figura 4 mostra como as expressões `3+4*5` e `(3+4)*5` são derivadas do diagrama de sintaxe.

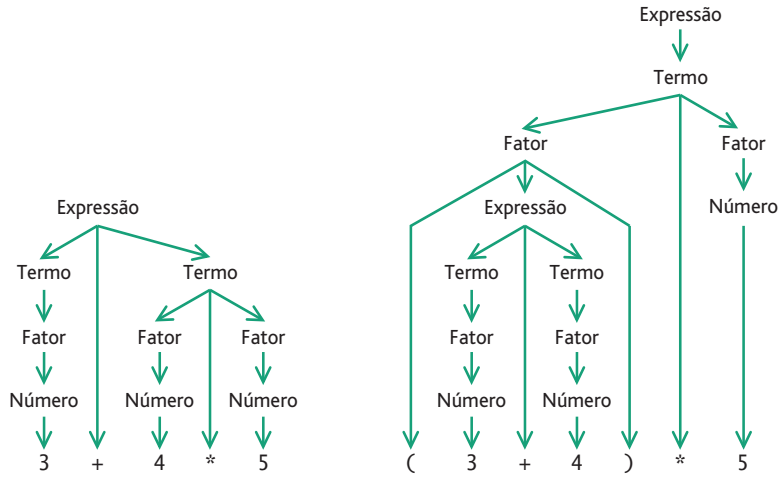


Figura 4 Árvores de sintaxe para duas expressões.

Por que os diagramas de sintaxe ajudam a calcular o valor da árvore? Se examinar as árvores de sintaxe, verá que elas representam exatamente as operações que devem ser executadas primeiro. Na primeira árvore, 4 e 5 devem ser multiplicados e então o resultado deve ser adicionado a 3. Na segunda árvore, 3 e 4 devem ser adicionados e o resultado dever ser multiplicado por 5.

No fim desta seção, você encontrará a implementação da classe `Evaluator`, que avalia essas expressões. O `Evaluator` usa uma classe `ExpressionTokenizer`, que divide uma string de entrada em tokens – números, operadores e parênteses. (Por questão de simplicidade, aceitamos apenas inteiros positivos como números e não permitimos espaços na entrada.)

Quando você chama `nextToken`, o próximo token de entrada é retornado como uma string. Também fornecemos outro método, `peekToken`, que permite ver o próximo token sem consumi-lo. Para ver por que o método `peekToken` é necessário, considere o diagrama de sintaxe *fator*. Se o próximo token for um `"*"` ou `"/"`, você quer continuar adicionando e subtraindo termos. Mas se o próximo token for outro caractere, como um `"+"` ou `"-"`, você quer parar sem realmente consumi-lo, para que o token possa ser considerado mais tarde.

Para calcular o valor de uma expressão, implementamos três métodos: `getExpressionValue`, `getTermValue` e `getFactorValue`. O método `getExpressionValue` primeiro chama `getTermValue` para obter o valor do primeiro termo da expressão. Em seguida, verifica se o próximo token de entrada é um de `+` ou `-`. Se for, chama `getTermValue` novamente e o adiciona ou subtrai.

```
public int getExpressionValue()
{
   int value = getTermValue();
   boolean done = false;
   while (!done)
   {
```

```
      String next = tokenizer.peekToken();
      if ("+".equals(next) || "-".equals(next))
      {
         tokenizer.nextToken(); // Descarta "+" ou "-"
         int value2 = getTermValue();
         if ("+".equals(next)) value = value + value2;
         else value = value - value2;
      }
      else done = true;
   }
   return value;
}
```

O método `getTermValue` chama `getFactorValue` da mesma maneira, multiplicando ou dividindo os valores de fator.

Por fim, o método `getFactorValue` verifica se a próxima entrada é um número, ou se ela começa ou não com um token `(`. No primeiro caso, o valor é simplesmente o valor do número. Entretanto, no segundo caso, o método `getFactorValue` faz uma chamada recursiva para `getExpressionValue`. Portanto, os três métodos são mutuamente recursivos.

```
public int getFactorValue()
{
   int value;
   String next = tokenizer.peekToken();
   if ("(".equals(next))
   {
      tokenizer.nextToken(); // Descarta"("
      value = getExpressionValue();
      tokenizer.nextToken(); // Descarta")"
   }
   else
      value = Integer.parseInt(tokenizer.nextToken());
   return value;
}
```

Para ver a recursão mútua claramente, rastreie a expressão `(3+4)*5`:

- `getExpressionValue` chama `getTermValue`
  - `getTermValue` chama `getFactorValue`
    - `getFactorValue` consome a entrada `(`
    - `getFactorValue` chama `getExpressionValue`
      - `getExpressionValue` retorna com o valor `7`, tendo consumido `3 + 4`. Essa é a chamada recursiva.
    - `getFactorValue` consome a entrada `)`
    - `getFactorValue` retorna `7`
  - `getTermValue` consome as entradas `*` e `5` e retorna `35`
- `getExpressionValue` retorna `35`

Como sempre, com uma solução recursiva você precisa assegurar que a recursão termine. Nessa situação, isso é fácil de ver. Se `getExpressionValue` chama a si próprio, a segunda chamada usa uma subexpressão mais curta que a expressão original. Em cada chamada recursiva, pelo menos alguns dos tokens da string de entrada são consumidos, então finalmente a recursão deve chegar a um fim.

ch13/expr/Evaluator.java

```
/**
   Uma classe que pode calcular o valor de uma expressão aritmética.
*/
public class Evaluator
{
   /**
      Constrói um avaliador.
      @param anExpression string contendo a expressão
      a ser avaliada
   */
   public Evaluator(String anExpression)
   {
      tokenizer = new ExpressionTokenizer(anExpression);
   }

   /**
      Avalia a expressão.
      @return valor da expressão
   */
   public int getExpressionValue()
   {
      int value = getTermValue();
      boolean done = false;
      while (!done)
      {
         String next = tokenizer.peekToken();
         if ("+".equals(next) || "-".equals(next))
         {
            tokenizer.nextToken(); // Descarta "+" ou "-"
            int value2 = getTermValue();
            if ("+".equals(next)) value = value + value2;
            else value = value - value2;
         }
         else done = true;
      }
      return value;
   }

   /**
      Avalia o próximo termo localizado na expressão.
      @return valor do termo
   */
   public int getTermValue()
   {
      int value = getFactorValue();
      boolean done = false;
      while (!done)
      {
         String next = tokenizer.peekToken();
```

```
50          if ("*".equals(next) || "/".equals(next))
51          {
52             tokenizer.nextToken();
53             int value2 = getFactorValue();
54             if ("*".equals(next)) value = value * value2;
55             else value = value / value2;
56          }
57          else done = true;
58       }
59       return value;
60    }
61
62    /**
63       Avalia o próximo fator localizado na expressão.
64       @return valor do fator
65    */
66    public int getFactorValue()
67    {
68       int value;
69       String next = tokenizer.peekToken();
70       if ("(".equals(next))
71       {
72          tokenizer.nextToken(); // Descarta "("
73          value = getExpressionValue();
74          tokenizer.nextToken(); // Descarta ")"
75       }
76       else
77          value = Integer.parseInt(tokenizer.nextToken());
78       return value;
79    }
80
81     private ExpressionTokenizer tokenizer;
82 }
```

ch13/expr/ExpressionTokenizer.java

```
1  /**
2     Essa classe divide uma string que descreve uma expressão
3     em tokens: números, parênteses e operadores.
4  */
5  public class ExpressionTokenizer
6  {
7     /**
8        Constrói um tokenizador (divide uma string em tokens).
9        @param anInput string a ser dividida em tokens
10    */
11    public ExpressionTokenizer(String anInput)
12    {
13       input = anInput;
14       start = 0;
15       end = 0;
16       nextToken();
17    }
18
```

```
19      /**
20         Observa o próximo token sem consumi-lo.
21         @return próximo token ou null se não houver mais tokens
22      */
23      public String peekToken()
24      {
25         if (start >= input.length()) return null;
26         else return input.substring(start, end);
27      }
28
29      /**
30         Obtém o próximo token e move o tokenizer para o token seguinte.
31         @return próximo token ou null se não houver mais tokens
32      */
33      public String nextToken()
34      {
35         String r = peekToken();
36         start = end;
37         if (start >= input.length()) return r;
38         if (Character.isDigit(input.charAt(start)))
39         {
40            end = start + 1;
41            while (end < input.length()
42                  && Character.isDigit(input.charAt(end)))
43               end++;
44         }
45         else
46            end = start + 1;
47         return r;
48      }
49
50      private String input;
51      private int start;
52      private int end;
53   }
```

ch13/expr/ExpressionCalculator.java

```
1   import java.util.Scanner;
2
3   /**
4      Esse programa calcula o valor de uma expressão
5      consistindo em números, operadores aritméticos e parênteses.
6   */
7   public class ExpressionCalculator
8   {
9      public static void main(String[] args)
10     {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter an expression: ");
13        String input = in.nextLine();
14        Evaluator e = new Evaluator(input);
15        int value = e.getExpressionValue();
16        System.out.println(input + "=" + value);
```

```
    }
}
```

Saída

```
Enter an expression: 3+4*5
3+4*5=23
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

9. Qual é a diferença entre um termo e um fator? Por que precisamos desses dois conceitos?
10. Por que o analisador de sintaxe de expressão usa recursão mútua?
11. O que acontece se você tentar analisar sintaticamente a expressão inválida `3+4*)5`? Especificamente, qual método lança uma exceção?

RESUMO DO CAPÍTULO

1. Um cálculo recursivo resolve um problema utilizando a solução do mesmo problema com valores mais simples.
2. Para uma recursão terminar, há casos especiais para os valores mais simples.
3. Às vezes é mais fácil encontrar uma solução recursiva se você fizer uma alteração sutil no problema original.
4. Ocasionalmente, uma solução recursiva executa muito mais lentamente do que sua correspondente iterativa. Entretanto, na maioria dos casos, a solução recursiva é apenas um pouco mais lenta.
5. Em muitos casos, uma solução recursiva é mais fácil de entender e implementar corretamente do que uma solução iterativa.
6. Em uma recursão mútua, um conjunto de métodos cooperativos chama um ao outro repetidamente.

EXERCÍCIOS DE REVISÃO

★ **Exercício R13.1.** Defina os termos

a. Recursão
b. Iteração
c. Recursão infinita
d. Método auxiliar recursivo

★★ **Exercício R13.2.** Esboce, mas não implemente, uma solução recursiva para localizar o menor valor em um array.

★★ **Exercício R13.3.** Esboce, mas não implemente, uma solução recursiva para classificar um array de números. *Dica:* primeiro localize o menor valor no array.

★★ **Exercício R13.4.** Esboce, mas não implemente, uma solução recursiva para gerar todos os subconjuntos do conjunto $\{1, 2, \ldots, n\}$.

★★★ **Exercício R13.5.** O Exercício P13.12 mostra uma maneira iterativa de gerar todas as permutações da seqüência $(0, 1, \ldots, n - 1)$. Explique por que o algoritmo produz o resultado correto.

★ **Exercício R13.6.** Escreva uma definição recursiva para x^n, onde $n \geq 0$, semelhante à definição recursiva dos números de Fibonacci. *Dica:* como calcular x^n de x^{n-1}? Como a recursão termina?

★★ **Exercício R13.7.** Aprimore o Exercício R13.6 computando x^n como $(x^{n/2})^2$ se n for par. Por que essa abordagem é significativamente mais rápida? (*Dica:* calcule x^{1023} e x^{1024} de ambas as maneiras.)

★ **Exercício R13.8.** Escreva uma definição recursiva para $n! = 1 \times 2 \times \ldots \times n$, semelhante à definição recursiva dos números de Fibonacci.

★★ **Exercício R13.9.** Descubra a freqüência com que a versão recursiva de `fib` chama a si própria. Conserve uma variável estática `fibCount` e incremente-a uma vez em cada chamada de `fib`. Qual é o relacionamento entre `fib(n)` e `fibCount`?

★★★ **Exercício R13.10.** Quantos movimentos são necessários no problema "Torres de Hanói" do Exercício P13.13 para mover n discos? *Dica:* como explicado no exercício,

$$\text{moves}(1) = 1$$
$$\text{moves}(n) = 2 \,.\, \text{moves}(n-1) + 1$$

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P13.1.** Escreva um método recursivo `void reverse()` que inverte uma frase. Por exemplo:

```
Sentence greeting = new Sentence("Hello!");
greeting.reverse();
System.out.println(greeting.getText());
```

imprime a string `"!olleH"`. Implemente uma solução recursiva removendo o primeiro caractere, invertendo uma frase com o texto restante e combinando os dois.

★★ **Exercício P13.2.** Refaça o Exercício P13.1 com um método auxiliar recursivo que inverte uma substring do texto de mensagem.

★ **Exercício P13.3.** Implemente o método `reverse` do Exercício P13.1 como uma iteração.

★★ **Exercício P13.4.** Utilize recursão para implementar um método `boolean find(String t)` que testa se uma string está ou não contida em uma frase:

```
Sentence s = new Sentence("Mississippi!");
boolean b = s.find("sip"); // Retorna true
```

Dica: se o texto começar com a string que você quer corresponder, então seu trabalho está pronto. Se não, considere a frase que você obtém removendo o primeiro caractere.

★★ **Exercício P13.5.** Utilize recursão para implementar um método `int indexOf(String t)` que retorna a posição inicial da primeira substring do texto que corresponde a `t`. Retorne –1 se `t` não for uma substring de `s`. Por exemplo,

```
Sentence s = new Sentence("Mississippi!");
int n = s.indexOf("sip"); // Retorna 6
```

Dica: esse problema é um pouco mais difícil do que o anterior, porque você deve monitorar até onde a correspondência ocorre desde o início da frase. Torne esse valor um parâmetro de um método auxiliar.

★ **Exercício P13.6.** Utilizando recursão, localize o maior elemento em um array.

```
public class DataSet
{
   public DataSet(int[] values, int first, int last) { . . . }
   public int getMaximum() { . . . }
   . . .
}
```

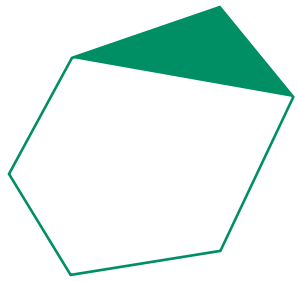
Dica: localize o maior elemento no subconjunto contendo todos, exceto o último elemento. Então compare esse valor máximo com o valor do último elemento.

★ **Exercício P13.7.** Utilizando recursão, calcule a soma de todos os valores em um array.

```
public class DataSet
{
   public DataSet(int[] values, int first, int last) { . . . }
   public int getSum() { . . . }
   . . .
}
```

★★ **Exercício P13.8.** Utilizando recursão, calcule a área de um polígono. Corte fora um triângulo e utilize o fato de que um triângulo com cantos (x_1, y_1), (x_2, y_2), (x_3, y_3) tem área:

$$\frac{|x_1 y_2 + x_2 y_3 + x_3 y_1 - y_1 x_2 - y_2 x_3 - y_3 x_1|}{2}$$



★★★ **Exercício P13.9.** Implemente um `SubstringGenerator` que gera todas as substrings de uma string. Por exemplo, as substrings da string `"rum"` são as sete strings:

```
"r", "ru", "rum", "u", "um", "m", ""
```

Dica: primeiro enumere todas as substrings que iniciam com o primeiro caractere. Há n delas se a string tiver comprimento n. Então enumere as substrings da string que você obtém removendo o primeiro caractere.

★★★ **Exercício P13.10.** Implemente um `SubsetGenerator` que gera todos os subconjuntos dos caracteres de uma string. Por exemplo, os subconjuntos dos caracteres da string `"rum"` são as oito strings:

```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```

Observe que os subconjuntos não precisam ser substrings – por exemplo, `"rm"` não é uma substring de `"rum"`.

★★★ **Exercício P13.11.** Neste exercício, você transformará o `PermutationGenerator` da Seção 13.2 (que calculou todas as permutações de uma vez) em um `PermutationIterator` (que calcula uma permutação por vez.)

```
public class PermutationIterator
{
   public PermutationIterator(String s) { . . . }
   public String nextPermutation() { . . . }
   public boolean hasMorePermutations() { . . . }
}
```

Eis como você imprimiria todas as permutações da string `"eat"`:

```
PermutationIterator iter = new PermutationIterator("eat");
while (iter.hasMorePermutations())
   System.out.println(iter.nextPermutation());
```

Agora precisamos de uma maneira de iterar pelas permutações recursivamente. Considere a string `"eat"`. Como antes, geraremos todas as permutações que iniciam com a letra

`'e'`, depois as que iniciam com `'a'` e, por fim, as que iniciam com `'t'`. Como geramos as permutações que iniciam com `'e'`? Crie outro objeto `PermutationIterator` (chamado `tailIterator`) que itera pelas permutações da substring `"at"`. No método `nextPermutation`, simplesmente pergunte para `tailIterator` qual é *sua* próxima permutação, e então adicione o `'e'` na frente. Mas há um caso especial. Quando `tailIterator` esgotar as permutações, todas as permutações que iniciam com a letra atual foram enumeradas. Então

- Incremente a posição atual.
- Calcule a substring que contém todas as letras exceto a atual.
- Crie um novo iterador de permutação para a substring.

Seu trabalho estará concluído quando a posição atual alcançar o fim da string.

★★★ **Exercício P13.12.** A seguinte classe gera todas as permutações dos números 0, 1, 2, . . ., $n - 1$, sem utilizar recursão.

```
public class NumberPermutationIterator
{
   public NumberPermutationIterator(int n)
   {
      a = new int[n];
      done = false;
      for (int i = 0; i < n; i++) a[i] = i;
   }

   public int[] nextPermutation()
   {
      if (a.length <= 1) return a;

      for (int i = a.length - 1; i > 0; i--)
      {
         if (a[i - 1] < a[i])
         {
            int j = a.length - 1;
            while (a[i - 1] > a[j]) j--;
            swap(i - 1, j);
            reverse(i, a.length - 1);
            return a;
         }
      }
      return a;
   }

   public boolean hasMorePermutations()
   {
      if (a.length <= 1) return false;
      for (int i = a.length - 1; i > 0; i--)
      {
         if (a[i - 1] < a[i]) return true;
      }
      return false;
   }
   public void swap(int i, int j)
   {
```

```
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }
   public void reverse(int i, int j)
   {
      while (i < j) { swap(i, j); i++; j--; }
   }
   private int[] a;
}
```

O algoritmo usa o fato de que o conjunto a ser permutado consiste em números distintos. Portanto, você não pode utilizar o mesmo algoritmo para calcular as permutações dos caracteres em uma string. Mas você pode utilizar essa classe para obter todas as permutações das posições de caractere e, então, calcular uma string cujo `i`-ésimo caractere é `word.charAt(a[i])`. Utilize essa abordagem para reimplementar o `PermutationIterator` do Exercício P13.11 sem recursão.

★★ **Exercício P13.13.** *Torres de Hanói.* Esse é um quebra-cabeça bem-conhecido. Uma pilha de discos com tamanho decrescente deve ser transportada da haste esquerda para a haste direita. A haste intermediária pode ser utilizada como armazenamento temporário (veja Figura 5). Pode-se mover um disco por vez, de qualquer haste para qualquer outra. Você pode colocar discos menores sobre discos maiores, não o contrário.

Escreva um programa que imprime as jogadas necessárias para resolver o quebra-cabeça para *n* discos. (Pergunte ao usuário o valor de *n* no início do programa.) Imprima as jogadas/movimentos na forma

```
Move disk from peg 1 to peg 3
```

Dica: implemente uma classe `DiskMover`. O construtor aceita

- A haste de origem a partir da qual mover os discos (1, 2 ou 3)
- A haste alvo para a qual mover os discos (1, 2 ou 3)
- O número de discos a mover

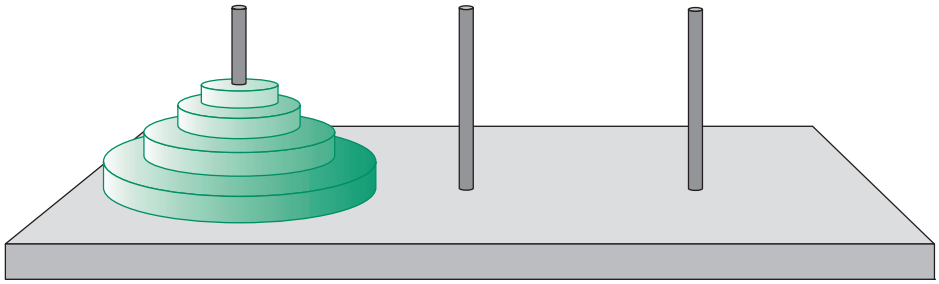


Figura 5 Torres de Hanói.

Um transportador de disco que move um único disco de uma haste para a outra simplesmente tem um método `nextMove` que retorna uma string

```
Move disk from peg origem to peg destino
```

Um transportador de disco com mais de um disco a mover deve trabalhar mais. Ele precisa de outro `DiskMover` para ajudá-lo. No construtor, crie um `DiskMover(source, other, disks - 1)` onde `other` é a haste diferente de `from` e `target`.

O `nextMove` pergunta a esse transportador de disco qual é seu próximo movimento até ele terminar. O efeito é mover os primeiros `disks - 1` discos para outra haste. Então o método `nextMove` emite um comando para mover um disco a partir da haste `from` para a haste `to`. Por fim, ele constrói outro transportador de disco `DiskMover(other, target, disks - 1)` que gera o movimento que desloca os discos da outra haste para a haste alvo.

Dica: isto ajuda a monitorar o estado do transportador de disco:

- `BEFORE_LARGEST`: o transportador auxiliar move a pilha menor para a outra haste.
- `LARGEST`: move o maior disco da fonte para o destino.
- `AFTER_LARGEST`: o transportador auxiliar move a menor pilha da outra haste para o alvo.
- `DONE`: todos os movimentos foram concluídos.

Teste seu programa da seguinte maneira:

```
DiskMover mover = new DiskMover(1, 3, n);
while (mover.hasMoreMoves())
   System.out.println(mover.nextMove());
```

★★★ **Exercício P13.14.** *Escapando de um labirinto*. Atualmente você está localizado dentro de um labirinto. As paredes do labirinto são indicadas por asteriscos (`*`).

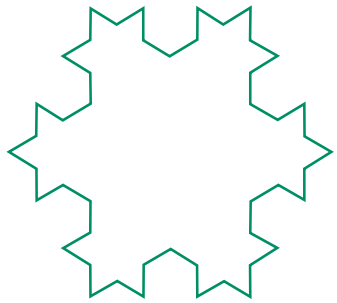
```
* *******
*     * *
* ***** *
* * *   *
* * *** *
*   *   *
*** * * *
*     * *
******* *
```

Utilize a seguinte abordagem recursiva para verificar se você pode escapar do labirinto: se você estiver em uma saída, retorne `true`. Verifique recursivamente se você pode escapar de uma das localizações vizinhas vazias sem visitar a localização atual. Esse método apenas testa se há um caminho para fora do labirinto. Você ganha um crédito extra se souber imprimir um caminho que leva a uma saída.

★★★G **Exercício P13.15.** *O floco de neve de Koch.* A forma de um floco de neve é definida recursivamente da seguinte maneira. Inicie com um triângulo equilateral:

Em seguida, aumente o tamanho por um fator de três e substitua cada linha reta por quatro segmentos de linha.

Repita o processo.



Escreva um programa que desenha as iterações dessa curva. Forneça um botão que, quando clicado, produz a próxima iteração.

★★ **Exercício P13.16.** A computação recursiva de números de Fibonacci pode ser significativamente acelerada monitorando-se os valores que já foram calculados. Forneça uma implementação do método `fib` que usa essa estratégia. Sempre que você retornar um novo valor, armazene-o também em um array auxiliar. Entretanto, antes de embarcar em um cálculo, consulte o array para localizar se o resultado já foi computado. Compare o tempo de execução de sua implementação aprimorada com aquela da implementação recursiva original e a implementação baseada em laços.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 13.1.** Aprimore o analisador de sintaxe de expressão da Seção 13.5 para tratar expressões mais sofisticadas, como expoentes, e funções matemáticas, como `sqrt` (raiz quadrada) ou `sin` (seno).

★★★G **Projeto 13.2.** Implemente uma versão gráfica do programa Torres de Hanói (veja o Exercício P13.13). Toda vez que o usuário clicar em um botão rotulado “Next”, desenhe o próximo movimento.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Suponha que omitíssemos a instrução. Ao calcular a área de um triângulo com largura 1, calculamos a área do triângulo com largura 0 como 0 e, então, adicionamos 1 para obter a área correta.
2. Você calcularia a menor área recursivamente, então retornaria `smallerArea + width + width - 1`.

```
[][][][]
[][][][]
[][][][]
[][][][]
```

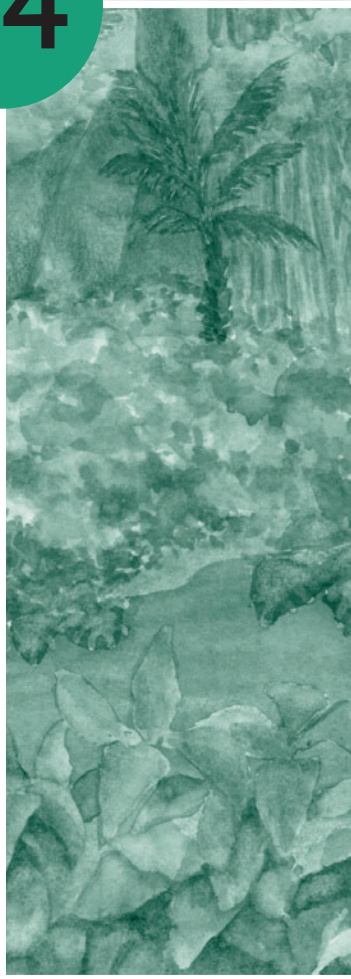
Naturalmente, seria mais simples calcular a área como `width * width`. Os resultados são idênticos porque

$$1 + 0 + 2 + 1 + 3 + 2 + \cdots + n + n - 1 = \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2.$$

3. Eles são `b` seguido pelas seis permutações de `eat`, `e` seguido pelas seis permutações de `bat`, `a` seguido pelas seis permutações de `bet`, e `t` seguido pelas seis permutações de `bea`.
4. Simplesmente altere `if (word.length() == 0)` para `if (word.length() <= 1)`, porque uma palavra de uma única letra também é sua única permutação.
5. Não – o primeiro poderia ter recebido um nome diferente como `substringIsPalindrome`.
6. Quando `start >= end`, isto é, quando a string investigada é vazia ou tem comprimento 1.
7. Não, a solução recursiva é quase tão eficiente quanto a abordagem iterativa. Ambas requerem $n - 1$ multiplicações para calcular $n!$.
8. Uma solução iterativa teria um laço cujo corpo calcula a próxima permutação a partir das anteriores. Mas não há nenhum mecanismo óbvio para obter a próxima permutação. Por exemplo, se você já localizou permutações `eat`, `eta` e `aet`, não é claro como você usa essas informações para obter a próxima permutação. De fato, há um mecanismo engenhoso para fazer exatamente isso, mas ele está longe de ser óbvio – veja o Exercício P13.12.
9. Os fatores são combinados por operadores multiplicativos (`*` e `/`), os termos são combinados por operadores aditivos (`+`, `–`). Precisamos de ambos para que a multiplicação tenha precedência sobre a adição.
10. Para tratar expressões entre parênteses, como `2+3*(4+5)`. A subexpressão `4+5` é tratada por uma chamada recursiva a `getExpressionValue`.
11. A chamada `Integer.parseInt` em `getFactorValue` lança uma exceção quando recebe a string `")"`.

Capítulo 14

Classificação e Pesquisa



Objetivos do capítulo

- Estudar vários algoritmos de classificação e pesquisa
- Perceber que algoritmos para a mesma tarefa podem ter grande diferença no desempenho
- Entender a notação O grande (big-Oh)
- Aprender a estimar e comparar o desempenho de algoritmos
- Aprender a medir o tempo de execução de um programa

Uma das tarefas mais comuns em processamento de dados é a classificação. Por exemplo, podemos precisar imprimir em ordem alfabética ou classificada por salário uma lista com grupo de funcionários. Estudaremos vários métodos de classificação neste capítulo e compararemos seu desempenho. Isso é, sem dúvida, um tratamento exaustivo do tópico classificação. É bem provável que você veja esse tópico novamente em seus estudos avançados de ciência da computação. Uma boa visão geral dos muitos métodos de classificação disponíveis pode ser encontrada em [1].

Uma vez que uma seqüência de objetos é classificada, é possível localizar objetos individuais rapidamente. Estudaremos o algoritmo de pesquisa binária, que realiza essa pesquisa rápida.

Conteúdo do Capítulo

14.1 Classificação por seleção 572

14.2 Medindo o tempo de execução do algoritmo de classificação por seleção 575

14.3 Analisando o desempenho do algoritmo de classificação por seleção 579

Tópico avançado 14.1: Classificação por inserção ⊕

Tópico avançado 14.2: O, Ômega e Teta ⊕

14.4 Classificação por intercalação 581

14.5 Analisando o algoritmo de classificação por intercalação 584

Tópico avançado 14.3: O algoritmo quicksort ⊕

Fato aleatório 14.1: A primeira programadora ⊕

14.6 Pesquisa 588

14.7 Pesquisa binária 590

14.8 Classificando dados reais 593

Erro comum 14.1: O método `compareTo` pode retornar qualquer inteiro, não só –1, 0 e 1 595

Tópico avançado 14.4: A interface `Comparable` parametrizada 595

Tópico avançado 14.5: A interface `Comparator` ⊕

14.1 Classificação por seleção

Nesta seção, mostraremos o primeiro dos vários algoritmos de classificação. Um *algoritmo de classificação* reorganiza os elementos de uma coleção de modo que sejam armazenados em ordem de classificação. Para manter os exemplos simples, discutiremos como classificar um array de inteiros antes de avançarmos para a classificação de strings ou dados mais complexos. Considere o seguinte array `a`:

11	9	17	5	12

O algoritmo de classificação por seleção classifica um array localizando repetidamente o menor elemento da região inferior não-classificada e deslocando-o para frente.

Um primeiro passo óbvio é localizar o menor elemento. Nesse caso o menor elemento é 5, armazenado em `a[3]`. Devemos mover o 5 para o início do array. Naturalmente, já existe um elemento armazenado em `a[0]`, a saber, 11. Portanto, não podemos simplesmente mover `a[3]` para `a[0]` sem mover o 11 para outro lugar. Ainda não sabemos onde o 11 deve acabar, mas sabemos com certeza que ele não deve estar em `a[0]`. Simplesmente o tiramos do caminho *permutando-o* com `a[3]`.

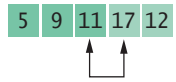
5	9	17	11	12

Agora o primeiro elemento está no lugar correto. Na figura anterior, a cor mais escura indica a parte do array que já está classificada.

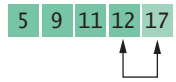
Em seguida, pegamos a menor das entradas restantes `a[1] . . . a[4]`. Esse valor mínimo, 9, já está no lugar certo. Nesse caso não precisamos fazer nada e podemos simplesmente estender a área classificada por uma à direita:

5	9	17	11	12

Repita o processo. O valor mínimo da região não-classificada é 11, que precisa ser permutado com o primeiro valor da região não-classificada, 17:



Agora a região não-classificada tem a largura de apenas dois elementos, mas mantemos a mesma estratégia bem-sucedida. O valor mínimo é 12 e o permutamos com o primeiro valor, 17.



Isso nos deixa com uma região não processada de comprimento 1, mas, naturalmente, uma região de comprimento 1 está sempre classificada. Terminamos essa parte.

Vamos programar esse algoritmo. Para esse programa, bem como para os outros deste capítulo, utilizaremos um método utilitário para gerar um array com entradas aleatórias. Colocamos esse array em uma classe `ArrayUtil` para não precisarmos repetir o código em cada exemplo. Para mostrar o array, chamamos o método `toString` estático da classe `Arrays` na biblioteca Java e imprimimos a string resultante.

Esse algoritmo classificará qualquer array de inteiros. Se a velocidade não fosse importante, ou se simplesmente não houvesse nenhum método de classificação melhor disponível, poderíamos parar por aqui a discussão sobre classificação. Como a próxima seção mostra, porém, esse algoritmo, embora completamente correto, apresenta um desempenho insatisfatório quando executado em um conjunto de dados grande.

O Tópico Avançado 14.1 discute a classificação por inserção, outro algoritmo de classificação simples (e igualmente ineficiente).

ch14/selsort/SelectionSorter.java

```
1  /**
2     Essa classe classifica um array, utilizando o
3     algoritmo de classificação por seleção.
4  */
5  public class SelectionSorter
6  {
7     /**
8        Constrói um classificador por seleção.
9        @param anArray array a ser classificado
10    */
11    public SelectionSorter(int[] anArray)
12    {
13       a = anArray;
14    }
15
16    /**
17       Classifica o array gerenciado por esse classificador por seleção.
18    */
19    public void sort()
20    {
21       for (int i = 0; i < a.length - 1; i++)
22       {
23          int minPos = minimumPosition(i);
```

```
24            swap(minPos, i);
25         }
26      }
27
28      /**
29         Localiza o menor elemento no intervalo inferior do array.
30         @param from primeira posição em a a comparar
31         @return posição do elemento menor no
32         intervalo a[from] . . . a[a.length - 1]
33      */
34      private int minimumPosition(int from)
35      {
36         int minPos = from;
37         for (int i = from + 1; i < a.length; i++)
38            if (a[i] < a[minPos]) minPos = i;
39         return minPos;
40      }
41
42      /**
43         Permuta duas entradas do array.
44         @param i primeira posição a ser permutada
45         @param j segunda posição a ser permutada
46      */
47      private void swap(int i, int j)
48      {
49         int temp = a[i];
50         a[i] = a[j];
51         a[j] = temp;
52      }
53
54      private int[] a;
55   }
```

ch14/selsort/SelectionSortDemo.java

```
1   import java.util.Arrays;
2
3   /**
4      Esse programa demonstra o algoritmo de classificação por seleção
5      classificando um array que é preenchido com números aleatórios.
6   */
7   public class SelectionSortDemo
8   {
9      public static void main(String[] args)
10     {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13
14        SelectionSorter sorter = new SelectionSorter(a);
15        sorter.sort();
16
17        System.out.println(Arrays.toString(a));
18     }
19  }
```

ch14/selsort/ArrayUtil.java

```
import java.util.Random;

/**
   Essa classe contém métodos utilitários para manipulação de arrays.
*/
public class ArrayUtil
{
   /**
      Cria um array preenchido com valores aleatórios.
      @param length comprimento do array
      @param n número de possíveis valores aleatórios
      @return array preenchido com length números entre
      0 e n - 1
   */
   public static int[] randomIntArray(int length, int n)
   {
      int[] a = new int[length];
      for (int i = 0; i < a.length; i++)
         a[i] = generator.nextInt(n);

      return a;
   }

     private static Random generator = new Random();
}
```

Saída típica

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Por que precisamos da variável `temp` no método `swap`? O que aconteceria se você simplesmente atribuísse `a[i]` a `a[j]` e `a[j]` a `a[i]`?
2. Que passos o algoritmo de classificação por seleção percorre para classificar a seqüência 6 5 4 3 2 1?

14.2 Medindo o tempo de execução do algoritmo de classificação por seleção

Para medir o desempenho de um programa, você poderia simplesmente executá-lo e medir quanto tempo leva usando um cronômetro. Entretanto, a maioria dos nossos programas executa com muita rapidez e, assim, não é fácil determinar exatamente seu tempo. Além disso, quando um programa gasta um tempo de execução considerável, uma porção desse tempo é utilizada apenas para carregar o programa do disco na memória (pelo que não devemos penalizá-lo) ou para saída em tela (cuja velocidade depende do modelo de computador, mesmo para computadores com velocidades de CPU idênticas. Em vez disso,

criaremos uma classe `StopWatch`. Essa classe funciona como um verdadeiro cronômetro. Você pode iniciá-lo, pará-lo e ler o tempo decorrido. A classe usa o método `System.currentTimeMillis`, que retorna os milissegundos que se passaram desde a zero hora de 1º de janeiro de 1970. Naturalmente, você não está preocupado com o número absoluto de segundos desde esse momento histórico, mas a *diferença* dessas duas contagens nos dá o número de milissegundos de um intervalo de tempo. Eis o código da classe `StopWatch`:

ch14/selsort/StopWatch.java

```
/**
   Um cronômetro acumula tempo enquanto está executando. Você pode
   iniciar e parar o cronômetro repetidamente. Você pode utilizar um
   cronômetro para medir o tempo de execução de um programa.
*/
public class StopWatch
{
   /**
      Constrói um cronômetro que está no estado parado
      e não tem tempo acumulado.
   */
   public StopWatch()
   {
      reset();
   }

   /**
      Inicia o cronômetro. O tempo começa a se acumular agora.
   */
   public void start()
   {
      if (isRunning) return;
      isRunning = true;
      startTime = System.currentTimeMillis();
   }

   /**
      Pára o cronômetro. O tempo pára de se acumular e é
      adicionado ao tempo decorrido.
   */
   public void stop()
   {
      if (!isRunning) return;
      isRunning = false;
      long endTime = System.currentTimeMillis();
      elapsedTime = elapsedTime + endTime - startTime;
   }

   /**
      Retorna o tempo total decorrido.
      @return tempo total transcorrido
   */
   public long getElapsedTime()
   {
      if (isRunning)
```

```
46         {
47            long endTime = System.currentTimeMillis();
48            return elapsedTime + endTime - startTime;
49         }
50         else
51            return elapsedTime;
52      }
53
54      /**
55         Pára o relógio e redefine o tempo transcorrido como 0.
56      */
57      public void reset()
58      {
59         elapsedTime = 0;
60         isRunning = false;
61      }
62
63      private long elapsedTime;
64      private long startTime;
65      private boolean isRunning;
66   }
```

Eis como utilizaremos o cronômetro para medir o desempenho do algoritmo de classificação:

ch14/selsort/SelectionSortTimer.java

```
1   import java.util.Scanner;
2
3   /**
4      Esse programa mede quanto tempo leva para classificar um
5      array de um tamanho especificado pelo usuário com o
6      algoritmo de classificação por seleção.
7   */
8   public class SelectionSortTimer
9   {
10      public static void main(String[] args)
11      {
12         Scanner in = new Scanner(System.in);
13         System.out.print("Enter array size: ");
14         int n = in.nextInt();
15
16         // Constrói array aleatório
17
18         int[] a = ArrayUtil.randomIntArray(n, 100);
19         SelectionSorter sorter = new SelectionSorter(a);
20
21         // Usa o cronômetro para medir o tempo da classificação por seleção
22
23         StopWatch timer = new StopWatch();
24
25         timer.start();
26         sorter.sort();
27         timer.stop();
```

```
28
29        System.out.println("Elapsed time: "
30              + timer.getElapsedTime() + " milliseconds");
31     }
32  }
```

Saída

```
Enter array size: 100000
Elapsed time: 27880 milliseconds
```

Começando a medir o tempo imediatamente antes da classificação e parando o cronômetro logo depois, você não conta o tempo que leva para inicializar o array ou o tempo que o programa espera até o usuário digitar `n`.

Eis os resultados de algumas execuções de exemplo:

n	Milissegundos
10.000	786
20.000	2.148
30.000	4.796
40.000	9.192
50.000	13.321
60.000	19.299

Essas medidas foram obtidas com um processador Pentium com uma velocidade de 2 GHz, executando Java 6 no sistema operacional Linux. Em outro computador os nú-

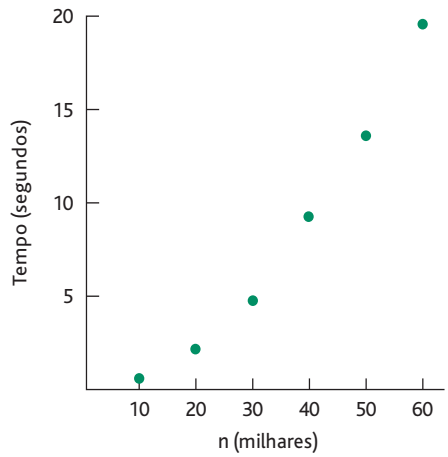


Figura 1 Tempo gasto pela classificação por seleção.

meros podem ser diferentes, mas a relação entre os números será a mesma. A Figura 1 mostra uma plotagem das medidas. Como você pode ver, dobrar o tamanho do conjunto de dados mais que dobra o tempo necessário para classificá-lo.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Aproximadamente quantos segundos levaria para classificar um conjunto de dados de 80.000 valores?
4. Examine o gráfico da Figura 1. Com que forma matemática isso se parece?

14.3 Analisando o desempenho do algoritmo de classificação por seleção

Vamos contar o número de operações que o programa deve executar para classificar um array com o algoritmo de classificação por seleção. Na verdade, não sabemos quantas operações de máquina são geradas para cada instrução Java nem quais dessas instruções são mais demoradas do que outras, mas podemos fazer uma simplificação. Contaremos apenas a freqüência com que um elemento do array é *visitado*. Cada visita exige quase a mesma quantidade de trabalho que outras operações, como incrementar subscritos e comparar valores.

Seja n o tamanho do array. Primeiro, devemos localizar o menor dos n números. Para alcançar isso, devemos visitar n elementos do array. Então permutamos os elementos, o que exige duas visitas. (Você pode argumentar que há certa probabilidade de não precisarmos permutar os valores. Isso é verdade, e pode-se refinar o cálculo para refletir essa observação. Como logo veremos, fazer isso não afetaria a conclusão geral.) No próximo passo, só precisamos visitar $n - 1$ elementos para localizar o mínimo. No passo a seguir, $n - 2$ elementos são visitados para localizar o mínimo. O último passo visita dois elementos para localizar o mínimo. Cada passo requer duas visitas para permutar os elementos. Portanto, o número total de visitas é

$$\begin{aligned} n + 2 + (n-1) + 2 + \cdots + 2 + 2 &= n + (n-1) + \cdots + 2 + (n-1) \cdot 2 \\ &= 2 + \cdots + (n-1) + n + (n-1) \cdot 2 \\ &= \frac{n(n+1)}{2} - 1 + (n-1) \cdot 2 \end{aligned}$$

porque:

$$1 + 2 + \cdots + (n-1) + n = \frac{n(n+1)}{2}$$

Depois de multiplicar e coletar termos de n, descobrimos que o número de visitas é

$$\tfrac{1}{2}n^2 + \tfrac{5}{2}n - 3$$

Obtemos uma equação quadrática em n. Isso explica por que o gráfico da Figura 1 parece um pouco com uma parábola.

Agora vamos simplificar ainda mais a análise. Quando você atribui um valor grande a n (por exemplo, 1.000 ou 2.000), então $\frac{1}{2}n^2$ será 500.000 ou 2.000.000. O termo menor, $\frac{5}{2}n - 3$, não contribui muito; é apenas 2.497 ou 4.997, uma gota no oceano se comparado às centenas de milhares ou ainda milhões de comparações especificadas pelo termo $\frac{1}{2}n^2$. Simplesmente ignoramos esses termos de nível inferior. Em seguida, ignoraremos o fator constante $\frac{1}{2}$. Não estamos interessados na contagem exata de visitas para um único n. Queremos comparar as proporções de contagens para valores diferentes de n. Por exemplo, podemos dizer que classificar um array de 2.000 números requer quatro vezes o número de visitas do que para classificar um array de 1.000 números:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

O fator $\frac{1}{2}$ impede comparações desse tipo. Simplesmente dizemos, "O número de visitas é da ordem n^2". Dessa maneira, podemos ver facilmente que o número de comparações aumenta quatro vezes quando o tamanho do array dobra: $(2n)^2 = 4n^2$.

Para indicar que o número de visitas é da ordem n^2, cientistas da computação costumam utilizar a *notação O grande (big-Oh):* O número de visitas é $O(n^2)$. Essa é uma abreviação conveniente.

Cientistas da computação utilizam a notação big-Oh $f(n) = O(g(n))$ para expressar que a função f não cresce mais rápido do que a função g.

Em geral, a expressão $f(n) = O(g(n))$ quer dizer que f não cresce mais rápido que g, ou, mais formalmente, que, para todo n maior que algum limiar, a relação $f(n)/g(n) \leq C$ para algum valor C constante. Em geral, a função g é escolhida por ser muito simples, como n^2 em nosso exemplo.

Para transformar uma expressão exata como

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

na notação O grande, simplesmente localize o termo de crescimento mais rápido, n^2, e ignore seu coeficiente constante, independentemente do quão grande ou pequeno ele possa ser.

Observamos antes que o número exato de operações de máquina e o número exato de microssegundos que o computador gasta neles são aproximadamente proporcionais ao número de visitas a elementos. Talvez haja umas 10 operações de máquina (incrementos, comparações, cargas de memória e armazenamentos) para cada visita a um elemento. O número de operações de máquina é então de cerca de $10 \times \frac{1}{2}n^2$. Novamente, não estamos interessados no coeficiente, então podemos dizer que o número de operações de máquina, e portanto o tempo gasto na classificação, é da ordem de n^2 ou $O(n^2)$.

A classificação por seleção é um algoritmo $O(n^2)$. Dobrar o conjunto de dados significa um aumento de quatro vezes no tempo de processamento.

O triste fato continua sendo que dobrar o tamanho do array quadruplica o tempo necessário para determiná-lo com a classificação por seleção. Quando o tamanho do array aumenta por um fator de 100, o tempo de classificação aumenta por um fator de 10.000. Classificar um array de um milhão de entradas, (por exemplo, para criar uma lista telefônica) leva 10.000 vezes mais tempo que classificar 10.000 entradas. Se 10.000 entradas podem ser classificadas em cer-

ca de 1/2 segundo (como em nosso exemplo), então classificar um milhão de entradas exige bem mais que uma hora. Veremos na próxima seção como aprimorar significativamente o desempenho do processo de classificação escolhendo um algoritmo mais sofisticado.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

5. Se você aumentar o tamanho de um conjunto de dados dez vezes, quanto tempo mais leva para classificá-lo com o algoritmo de classificação por seleção?
6. Qual é o tamanho que n precisa ter para que $\frac{1}{2}n^2$ seja maior que $\frac{5}{2}n - 3$?

TÓPICO AVANÇADO 14.1

Classificação por inserção

O Tópico Avançado 14.1 descreve a classificação por inserção, outro algoritmo de classificação simples que é comumente utilizado para arrays pequenos. Como a classificação por seleção, seu tempo de execução é $O(n^2)$.

TÓPICO AVANÇADO 14.2

O, Ômega e Teta

O Tópico Avançado 14.2 define as notações big-Teta e big-Ômega que descrevem o crescimento de uma função mais precisamente do que a notação big-Oh.

14.4 Classificação por intercalação

Nesta seção, veremos o algoritmo de classificação por intercalação, um algoritmo muito mais eficiente do que o da classificação por seleção. A idéia básica por trás da classificação por intercalação é muito simples.

Suponha que tivéssemos um array de 10 inteiros. Vamos nos iludir um pouco e esperar que a primeira metade do array já esteja perfeitamente classificada e de que a segunda metade também, como a seguir:

5	9	10	12	17	1	8	11	20	32

Agora é simples *intercalar* os dois arrays classificados em um array classificado, tirando um novo elemento do primeiro ou do segundo subarray e escolhendo o menor dos elementos a cada vez:

5	9	10	12	17	1	8	11	20	32	1									
5	9	10	12	17	1	8	11	20	32	1	5								
5	9	10	12	17	1	8	11	20	32	1	5	8							
5	9	10	12	17	1	8	11	20	32	1	5	8	9						
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10					
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11				
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12			

5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17		
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17	20	
5	9	10	12	17	1	8	11	20	32	1	5	8	9	10	11	12	17	20	32

Na verdade, provavelmente você já fez esse tipo de intercalação antes, quando você e um amigo precisaram classificar uma pilha de papéis. Vocês dividiram a pilha ao meio, cada um classificando sua parte e você intercalou os resultados depois.

O algoritmo de classificação por intercalação classifica um array dividindo-o ao meio, classificando recursivamente cada metade e, então intercalando as metades classificadas.

Isso é excelente, mas parece não resolver o problema para o computador. Ele ainda deve classificar a primeira e a segunda metade do array, pois não sabe muito bem pedir ajuda para alguns amigos. Mas acontece que se o computador continuar dividindo o array em subarrays cada vez menores, classificando todas as metades e intercalando-as, ele precisará de um número de passos significativamente menor do que na classificação por seleção.

Vamos escrever uma classe `MergeSorter` que implementa essa idéia. Quando `MergeSorter` classifica um array, ela cria dois arrays, cada um com metade do tamanho do original e os classifica recursivamente. Então ela intercala os dois arrays classificados:

```
public void sort()
{
   if (a.length <= 1) return;
   int[] first = new int[a.length / 2];
   int[] second = new int[a.length - first.length];
   System.arraycopy(a, 0, first, 0, first.length);
   System.arraycopy(a,
      first.length, second, 0, second.length);
   MergeSorter firstSorter = new MergeSorter(first);
   MergeSorter secondSorter = new MergeSorter(second);
   firstSorter.sort();
   secondSorter.sort();
   merge(first, second);
}
```

O método `merge` é sem graça, mas bem simples e direto. Você verá isso no código a seguir.

ch14/mergesort/MergeSorter.java

```
1  /**
2     Essa classe classifica um array, utilizando o algoritmo de classificação por intercalação.
3  */
4  public class MergeSorter
5  {
6     /**
7        Constrói um classificador por intercalação.
8        @param anArray array a ser classificado
9     */
10    public MergeSorter(int[] anArray)
11    {
12       a = anArray;
13    }
14
```

```
15      /**
16         Classifica o array gerenciado por esse classificador, por intercalação.
17      */
18      public void sort()
19      {
20         if (a.length <= 1) return;
21         int[] first = new int[a.length / 2];
22         int[] second = new int[a.length - first.length];
23         System.arraycopy(a, 0, first, 0, first.length);
24         System.arraycopy(a, first.length, second, 0, second.length);
25         MergeSorter firstSorter = new MergeSorter(first);
26         MergeSorter secondSorter = new MergeSorter(second);
27         firstSorter.sort();
28         secondSorter.sort();
29         merge(first, second);
30      }
31
32      /**
33         Intercala dois arrays classificados no array gerenciado por esse
34         classificador por intercalação.
35         @param first primeiro array classificado
36         @param second segundo array classificado
37      */
38      private void merge(int[] first, int[] second)
39      {
40         // Intercala as duas metades no array temporário
41
42         int iFirst = 0;
43            // Próximo elemento a considerar no primeiro array
44         int iSecond = 0;
45            // Próximo elemento a considerar no segundo array
46         int j = 0;
47            // Próxima posição aberta em a
48
49         // Contanto que nem iFirst nem iSecond tenham passado do fim, move
50         // o menor elemento para a
51         while (iFirst < first.length && iSecond < second.length)
52         {
53            if (first[iFirst] < second[iSecond])
54            {
55               a[j] = first[iFirst];
56               iFirst++;
57            }
58            else
59            {
60               a[j] = second[iSecond];
61               iSecond++;
62            }
63            j++;
64         }
65
66         // Observe que somente uma das duas chamadas a arraycopy a seguir
67         // copia entradas
68
```

```
      // Copia todas as entradas restantes do primeiro array
      System.arraycopy(first, iFirst, a, j, first.length - iFirst);

      // Copia todas as entradas restantes da segunda metade
      System.arraycopy(second, iSecond, a, j, second.length - iSecond);
   }

   private int[] a;
}
```

ch14/mergesort/MergeSortDemo.java

```
import java.util.Arrays;

/**
   Esse programa demonstra o algoritmo de classificação por intercalação
   classificando um array que é preenchido com números aleatórios.
*/
public class MergeSortDemo
{
   public static void main(String[] args)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));
      MergeSorter sorter = new MergeSorter(a);
      sorter.sort();
      System.out.println(Arrays.toString(a));
   }
}
```

Saída típica

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2, 76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62, 70, 76, 76, 81, 89, 90, 98]
```

Autoverificação da aprendizagem

7. Por que somente uma das duas chamadas `arraycopy` no fim do método `merge` é executada?
8. Execute manualmente o algoritmo de classificação por intercalação no array 8 7 6 5 4 3 2 1.

14.5 Analisando o algoritmo de classificação por intercalação

O algoritmo de classificação por intercalação parece muito mais complicado do que o de classificação por seleção, e é bem provável que a execução dessas repetidas subdivisões leve muito mais tempo. Entretanto, os resultados de medição dos tempos da classificação por intercalação parecem muito melhores do que os da classificação por seleção (veja a tabela a seguir).

n	Classificação por intercalação (milissegundos)	Classificação por seleção (milissegundos)
10.000	40	786
20.000	73	2.148
30.000	134	4.796
40.000	170	9.192
50.000	192	13.321
60.000	205	19.299

A Figura 2 mostra um gráfico comparando os dois conjuntos de dados de desempenho. Isso representa uma melhoria significativa. Para entender o porquê, vamos estimar o número de visitas a elementos do array que são necessárias para classificar um array com o algoritmo de classificação por intercalação. Primeiro, vamos abordar o processo de intercalação que ocorre depois que a primeira e a segunda metade são classificadas.

Cada passo no processo de intercalação adiciona mais um elemento a `a`. Esse elemento pode vir de `first` ou `second` e, na maioria dos casos, os elementos das duas metades devem ser comparados para ver qual deve ser selecionado. Vamos contar isso como 3 visitas (uma para `a`, uma para `first` e uma para `second`) por elemento, ou o total de $3n$ visitas, onde n indica o comprimento de `a`. Além disso, no início, precisamos copiar de `a` para `first` e `second`, resultando em outras $2n$ visitas, para um total de $5n$.

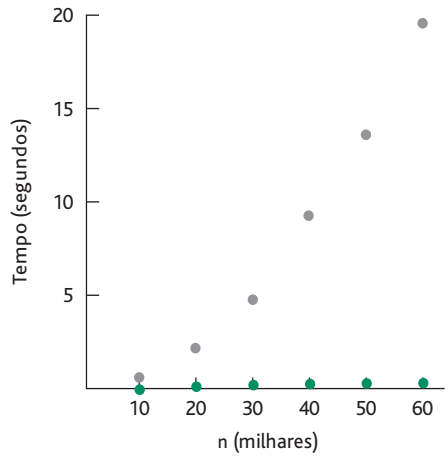


Figura 2 Tempos da classificação por intercalação (verde) *versus* por seleção (cinza).

Seja $T(n)$ o número de visitas necessárias para classificar um conjunto de n elementos pelo processo de classificação por intercalação, obteremos

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5n$$

porque classificar cada metade requer $T(n/2)$ visitas. De fato, se n não for par, então temos um subarray de tamanho $(n - 1)/2$ e um de tamanho $(n + 1)/2$. Embora constatássemos que esse detalhe não afeta o resultado da computação, assumiremos por ora que n é uma potência de 2, digamos $n = 2^m$. Dessa maneira, todos os subarrays podem ser igualmente divididos em duas partes.

Infelizmente, a fórmula

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n$$

não diz claramente a relação entre n e $T(n)$. Para entender a relação, vamos avaliar $T(n/2)$, utilizando a mesma fórmula

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 5\frac{n}{2}$$

Portanto

$$T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n$$

Vamos fazer isso novamente:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 5\frac{n}{4}$$

de onde

$$T(n) = 2 \times 2 \times 2T\left(\frac{n}{8}\right) + 5n + 5n + 5n$$

Isso generaliza de 2, 4, 8, a potências arbitrárias de 2:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 5nk$$

Lembre-se de que assumimos que $n = 2^m$; daí, para $k = m$,

$$\begin{aligned} T(n) &= 2^m T\left(\frac{n}{2^m}\right) + 5nm \\ &= nT(1) + 5nm \\ &= n + 5n \log_2(n) \end{aligned}$$

Como $n = 2^m$, temos $m = \log_2(n)$.

Para estabelecer a ordem de crescimento, tiramos o termo de ordem inferior n e ficamos com $5n \log_2(n)$. Descartamos o fator constante 5. Também é comum descartar a base do logaritmo, porque todos os logaritmos são relacionados por um fator constante. Por exemplo,

$$\log_2(x) = \log_{10}(x)/\log_{10}(2) \approx \log_{10}(x) \times 3.32193$$

Daí dizermos que a classificação por intercalação é um algoritmo $O(n \log(n))$.

A classificação por intercalação é um algoritmo $O(n \log(n))$. A função $n \log(n)$ cresce muito mais lentamente que n^2.

O algoritmo de classificação por intercalação é melhor do que o algoritmo de classificação por seleção $O(n^2)$? Pode apostar que sim. Lembre-se de que ele levou $100^2 = 10.000$ vezes mais para classificar um milhão de registros do que para classificar 10.000 registros com o algoritmo $O(n^2)$. Com o algoritmo $O(n \log(n))$, a relação é:

$$\frac{1.000.000 \log(1.000.000)}{10.000 \log(10.000)} = 100\left(\frac{6}{4}\right) = 150$$

Suponha por ora que essa classificação por intercalação dure o mesmo tempo que a classificação por seleção para um array de 10.000 inteiros, isto é, 3/4 de um segundo na máquina de teste. (Na verdade, é muito mais rápido do que isso.) Então levaria cerca de $0{,}75 \times 150$ segundos, ou menos de 2 minutos, para classificar um milhão de inteiros. Contraste isso com a classificação por seleção, que levaria mais ou menos 2 horas para executar a mesma tarefa. Como você pode ver, mesmo que você leve várias horas para aprender um algoritmo melhor, esse pode ser um tempo bem gasto.

Neste capítulo mal arranhamos a superfície desse interessante tópico. Há muitos algoritmos de classificação, alguns com desempenho ainda melhor do que o algoritmo de classificação por intercalação e a análise desses algoritmos pode ser um grande desafio. Se você é estudante de ciência da computação, é provável que reveja essas importantes questões em aulas de ciência da computação mais avançadas.

A classe `Arrays` implementa um método de classificação que você deve utilizar para seus programas Java.

Entretanto, ao escrever programas Java, você não precisa implementar seu próprio algoritmo de classificação. A classe `Arrays` contém métodos `sort` estáticos para classificar arrays de inteiros e números de ponto flutuante. Por exemplo, você pode classificar um array de inteiros simplesmente como

```
int[] a = . . .;
Arrays.sort(a);
```

Esse método `sort` usa o algoritmo *quicksort* – veja Tópico Avançado 14.3 para obter mais informações sobre esse algoritmo.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

9. Considerando os dados de medição de tempo do algoritmo de classificação por intercalação da tabela no começo desta seção, quanto tempo levaria para classificar um array de 100.000 valores?
10. Suponha que você tivesse um array `double[] values` em um programa Java. Como você o classificaria?

Tópico avançado 14.3

O algoritmo quicksort

O Tópico Avançado 14.3 descreve o algoritmo quicksort, um algoritmo muito utilizado que apresenta uma vantagem sobre a classificação por intercalação, porque nenhum array temporário é necessário para classificar e intercalar os resultados parciais. Em média, o algoritmo quicksort é um algoritmo $O(n \log(n))$. Como é mais simples, ele executa mais rapidamente do que a classificação por intercalação na maioria dos casos. Entretanto, seu comportamento de tempo de execução no pior cenário é $O(n^2)$.

Fato aleatório 14.1

A primeira programadora

O Fato Aleatório 14.1 conta a história de Charles Babbage, o criador de uma calculadora mecânica programável, e Ada Lovelace, sua amiga e patrocinadora. Lovelace é considerada por muitos como a primeira programadora do mundo.

14.6 Pesquisa

Suponha que você precise localizar o número de telefone de seu amigo. Você pesquisa seu nome na lista telefônica e, é claro, pode localizá-lo rapidamente, porque ela está classificada em ordem alfabética. É bem possível que você nunca tenha pensado na importância de a lista estar classificada. Para ver isso, pense no seguinte problema: suponha que você tenha um número de telefone e precise saber de quem ele é. É claro que você poderia ligar para esse número, mas suponha que, do outro lado da linha, ninguém atenda. Você poderia examinar a lista, um número por vez, até encontrá-lo. Isso seria obviamente um trabalho insano e você precisaria estar desesperado para tentar isso.

Essa experiência imaginária mostra a diferença entre uma pesquisa em um conjunto de dados não classificado e uma pesquisa em um conjunto de dados classificado. As duas próximas seções analisarão a diferença formalmente.

Uma pesquisa linear examina todos os valores em um array até localizar uma correspondência ou alcançar o fim.

Se você quiser localizar um número em uma seqüência de valores que ocorre em ordem arbitrária, não há nada que possa fazer para acelerar a pesquisa. Você deve simplesmente examinar cuidadosamente todos os elementos até localizar uma correspondência ou até alcançar o fim. Isso é chamado pesquisa linear ou seqüencial.

Uma pesquisa linear localiza um valor em um array em $O(n)$ passos.

Quanto tempo leva uma pesquisa linear? Se assumirmos que o elemento `v` está presente no array `a`, então a pesquisa média visita $n/2$ elementos, onde n é o comprimento do array. Se não estiver presente, então todos os n elementos devem ser inspecionados para verificar a ausência. De qualquer maneira, uma pesquisa linear é um algoritmo $O(n)$.

Eis uma classe que realiza pesquisas lineares em um array `a` de inteiros. Ao procurar pelo valor `v`, o método `search` retorna o primeiro índice da correspondência, ou `-1` se `v` não ocorrer em `a`.

ch14/linsearch/LinearSearcher.java

```
1  /**
2     Uma classe para executar pesquisas lineares por um array.
3  */
4  public class LinearSearcher
5  {
6     /**
7        Constrói o LinearSearcher.
8        @param anArray array de inteiros
9     */
10    public LinearSearcher(int[] anArray)
11    {
12       a = anArray;
13    }
14
15    /**
16       Localiza um valor em um array, utilizando o alogaritmo de pesquisa linear
17
18       @param v valor a pesquisar
19       @return índice em que o valor ocorre, ou -1
20       se não ocorrer no array
21    */
22    public int search(int v)
23    {
24       for (int i = 0; i < a.length; i++)
25       {
26          if (a[i] == v)
27             return i;
28       }
29       return -1;
30    }
31
32    private int[] a;
33 }
```

ch14/linsearch/LinearSearchDemo.java

```
1  import java.util.Arrays;
2  import java.util.Scanner;
3
4  /**
5     Esse programa demonstra o algoritmo de pesquisa linear.
6  */
7  public class LinearSearchDemo
8  {
9     public static void main(String[] args)
10    {
11       int[] a = ArrayUtil.randomIntArray(20, 100);
12       System.out.println(Arrays.toString(a));
```

```
13         LinearSearcher searcher = new LinearSearcher(a);
14
15         Scanner in = new Scanner(System.in);
16
17         boolean done = false;
18         while (!done)
19         {
20            System.out.print("Enter number to search for, -1 to quit: ");
21            int n = in.nextInt();
22            if (n == -1)
23               done = true;
24            else
25            {
26               int pos = searcher.search(n);
27               System.out.println("Found in position " + pos);
28            }
29         }
30      }
31   }
```

Saída típica

```
[46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85, 61, 88, 29, 65, 83, 88, 45, 88]
Enter number to search for, -1 to quit: 11
Found in position 8
```

Autoverificação da aprendizagem

11. Suponha que você precise examinar 1.000.000 registros para localizar um número de telefone. Quantos registros você espera pesquisar antes de localizar o número?
12. Por que você não pode utilizar um laço "for each" para `(int element : a)` no método `search`?

14.7 Pesquisa binária

Agora vamos procurar um item em uma seqüência de dados que foi anteriormente classificada. Naturalmente, ainda poderíamos fazer uma pesquisa linear, mas acontece que podemos fazer algo muito melhor que isso.

Considere o seguinte array classificado `a`. O conjunto de dados é:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Gostaríamos de verificar se o valor 15 está no conjunto de dados. Vamos restringir nossa pesquisa localizando se o valor está na primeira ou na segunda metade do array. O último ponto na primeira metade do conjunto de dados, `a[3]`, é 9, que é menor do que o valor que estamos procurando. Por isso precisamos procurar uma correspondência na segunda metade do array, isto é, na seqüência:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Agora o último valor da primeira metade dessa seqüência é 17; portanto, o valor deve ser localizado na seqüência:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

O último valor da primeira metade dessa seqüência muito curta é 12, que é menor que o valor que estamos procurando, então devemos procurar na segunda metade:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	5	8	9	12	17	20	32

Uma pesquisa binária localiza um valor em um array classificado determinando se o valor ocorre na primeira ou na segunda metade, repetindo a pesquisa em uma das metades.

É trivial ver que não temos uma correspondência, porque 15 ≠ 17. Se quiséssemos inserir 15 na seqüência, precisaríamos inseri-lo imediatamente antes de `a[5]`.

Esse processo de pesquisa é chamado de pesquisa binária, pois dividimos o tamanho da pesquisa ao meio em cada passo. Esse corte ao meio funciona somente porque sabemos que a seqüência de valores está classificada.

A classe a seguir implementa pesquisas binárias em um array classificado de inteiros. O método `search` retorna a posição da correspondência se a pesquisa for bem-sucedida, ou `-1` se `v` não for localizado em `a`.

ch14/binsearch/BinarySearcher.java

```
1   /**
2      Uma classe para executar pesquisas binárias por um array.
3   */
4   public class BinarySearcher
5   {
6      /**
7         Constrói um BinarySearcher.
8         @param anArray array classificado de inteiros
9      */
10     public BinarySearcher(int[] anArray)
11     {
12        a = anArray;
13     }
14
15     /**
16        Localiza um valor em um array classificado, utilizando o
17        algoritmo de pesquisa binário.
18        @param v valor a pesquisar
19        @return índice em que o valor ocorre, ou -1
20        se não ocorrer no array
21     */
```

```
22      public int search(int v)
23      {
24         int low = 0;
25         int high = a.length - 1;
26         while (low <= high)
27         {
28            int mid = (low + high) / 2;
29            int diff = a[mid] - v;
30
31            if (diff == 0) // a[mid] == v
32               return mid;
33            else if (diff < 0) // a[mid] < v
34               low = mid + 1;
35            else
36               high = mid - 1;
37         }
38         return -1;
39      }
40
41      private int[] a;
42   }
```

Vamos determinar o número de visitas a elementos do array necessárias para executar uma pesquisa. Podemos utilizar a mesma técnica empregada na análise de classificação por intercalação. Como examinamos o elemento do meio, que conta como uma comparação, e, então, pesquisamos o subarray esquerdo ou direito, temos

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Utilizando a mesma equação,

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

Inserindo esse resultado na equação original, obtemos

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

Isso pode ser generalizado como

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

Como ocorreu na análise de classificação por intercalação, fazemos a suposição simplificadora de que n é uma potência de 2, $n = 2^m$, onde $m = \log_2(n)$. Então obtemos

$$T(n) = 1 + \log_2(n)$$

Portanto, pesquisa binária é um algoritmo $O(\log(n))$.

Uma pesquisa binária localiza um valor em um array em $O(\log(n))$ passos.

Esse resultado tem um sentido intuitivo. Suponha que n é 100. Então, depois de cada pesquisa, o tamanho do intervalo de pesquisa é dividido ao meio, para 50, 25, 12, 6, 3 e 1. Depois de sete comparações você conclui essa operação. Isso concorda com nossa fórmula, porque $\log_2(100) \approx 6{,}64386$ e, de fato, a próxima maior potência de 2 é $2^7 = 128$.

Como uma pesquisa binária é muito mais rápida do que uma linear, vale a pena classificar um array primeiro e depois usar uma pesquisa binária? Depende. Se você pesquisar o array apenas uma vez, então é mais eficiente pagar por uma pesquisa linear $O(n)$ do que por uma classificação $O(n \log(n))$ e uma pesquisa binária $O(\log(n))$. Mas se você estiver fazendo muitas pesquisas no mesmo array, então a classificação é certamente vantajosa.

A classe `Arrays` contém um método `binarySearch` estático que implementa o algoritmo de pesquisa binária, mas com um aprimoramento útil. Se um valor não for localizado no array, então o valor retornado não é –1, mas $-k - 1$, onde k é a posição antes da qual o elemento deveria ser inserido. Por exemplo,

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
   // Retorna -3; v deve ser inserido antes da posição 2
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

13. Suponha que você precise examinar um array classificado com 1.000.000 de elementos para localizar um valor. Utilizando o algoritmo de pesquisa binária, quantos registros você espera pesquisar antes de encontrar o valor?
14. Por que é útil que o método `Arrays.binarySearch` indique a posição onde um elemento ausente deveria ser inserido?
15. Por que `Arrays.binarySearch` retorna $-k - 1$ e não $-k$ para indicar que um valor não está presente e deveria ser inserido antes da posição k?

14.8 Classificando dados reais

Neste capítulo estudamos como pesquisar e classificar arrays de inteiros. Naturalmente, em programas aplicativos, raramente há necessidade de pesquisar por uma coleção de inteiros. Entretanto, é fácil modificar essas técnicas para pesquisar por dados reais.

O método `sort` da classe `Arrays` classifica objetos de classes que implementam a interface `Comparable`.

A classe `Arrays` fornece um método `sort` estático para classificar arrays de objetos. Entretanto, a classe `Arrays` não pode saber como comparar objetos arbitrários. Suponha, por exemplo, que você tenha um array de objetos `Coin`. Não é óbvio como as moedas devem ser classificadas. Você poderia classificá-las por nome ou por valor. O método `Arrays.sort` não pode tomar essa decisão para você. Em vez disso, ele requer que os objetos pertençam a classes que implementam a interface `Comparable`. Essa interface tem um único método:

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

A chamada

```
a.compareTo(b)
```

deve retornar um número negativo se `a` precisar vir antes de `b`, 0 se `a` e `b` forem o mesmo e um número positivo, caso contrário.

Várias classes na biblioteca Java padrão, como as classes `String` e `Date`, implementam a interface `Comparable`.

Você também pode implementar a interface `Comparable` para suas próprias classes. Por exemplo, para classificar uma coleção de moedas, a classe `Coin` precisaria implementar essa interface e definir um método `compareTo`:

```
public class Coin implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
      Coin other = (Coin) otherObject;
      if (value < other.value) return -1;
      if (value == other.value) return 0;
      return 1;
   }
   . . .
}
```

Ao implementar o método `compareTo` da interface `Comparable`, você deve certificar-se de que o método define um *relacionamento* de ordenação total, com as três propriedades a seguir:

- *Anti-simétrica:* se `a.compareTo(b)` ≤ 0, então `b.compareTo(a)` ≥ 0
- *Reflexiva:* `a.compareTo(a)` $= 0$
- *Transitiva:* se `a.compareTo(b)` ≤ 0 e `b.compareTo(c)` ≤ 0, então `a.compareTo(c)` ≤ 0

Uma vez que sua classe `Coin` implementa a interface `Comparable`, você pode simplesmente passar um array de moedas para o método `Arrays.sort`:

```
Coin[] coins = new Coin[n];
// Adiciona moedas
. . .
Arrays.sort(coins);
```

A classe `Collections` contém um método `sort` que pode classificar listas de array.

Se as moedas são armazenadas em uma `ArrayList`, utilize o método `Collections.sort`; ele usa o algoritmo de classificação por intercalação:

```
ArrayList<Coin> coins = new ArrayList<Coin>();
// Adiciona moedas
. . .
Collections.sort(coins);
```

Como uma questão prática, você deve utilizar os métodos de classificação e pesquisa nas classes `Arrays` e `Collections`, e não aqueles que você mesmo escreve. Os algoritmos da biblioteca foram inteiramente depurados e otimizados. Portanto, o principal objetivo deste capítulo não foi ensiná-lo a implementar os algoritmos práticos de classificação e

pesquisa. Em vez disso, você aprendeu algo mais importante, a saber, que os algoritmos diferentes podem variar significativamente em desempenho e que vale a pena aprender mais sobre o projeto e análise de algoritmos.

AUTOVERIFICAÇÃO DA APRENDIZAGEM

16. Por que o método `Arrays.sort` não pode classificar um array de objetos `Rectangle`?
17. Que passos você precisaria seguir para classificar um array de objetos `BankAccount` aumentando o saldo?

ERRO COMUM 14.1

O método `compareTo` pode retornar qualquer inteiro, não só –1, 0 e 1

A chamada `a.compareTo(b)` tem permissão de retornar *qualquer* inteiro negativo para indicar que `a` deve vir antes de `b`, não necessariamente o valor –1. Isto é, o teste

```
if (a.compareTo(b) == -1) // ERRO!
```

geralmente está errado. Em vez disso, você deve testar

```
if (a.compareTo(b) < 0) // OK
```

Por que um método `compareTo` iria querer retornar um número diferente de –1, 0 ou 1? Às vezes, é conveniente apenas retornar a diferença de dois inteiros. Por exemplo, o método `compareTo` da classe `String` compara caracteres em posições correspondentes:

```
char c1 = charAt(i);
char c2 = other.charAt(i);
```

Se os caracteres forem diferentes, então o método simplesmente retorna sua diferença:

```
if (c1 != c2) return c1 - c2;
```

Essa diferença é um número negativo se `c1` for menor do que `c2`, mas não necessariamente é o número –1.

TÓPICO AVANÇADO 14.4

A interface `Comparable` parametrizada

Em relação à versão Java 5.0, a interface `Comparable` é um tipo parametrizado, semelhante ao tipo `ArrayList`:

```
public interface Comparable<T>
{
   int compareTo(T other)
}
```

O parâmetro de tipo especifica o tipo dos objetos que essa classe aceita para comparação. Normalmente, esse tipo é o mesmo tipo da classe. Por exemplo, a classe `Coin` implementaria `Comparable<Coin>`, assim:

```
public class Coin implements Comparable<Coin>
{
   . . .
   public int compareTo(Coin other)
   {
      if (value < other.value) return -1;
      if (value == other.value) return 0;
      return 1;
   }
   . . .
}
```

O parâmetro de tipo tem uma vantagem significativa: você não precisa utilizar uma coerção, ou typecasting, para converter um parâmetro `Object` no tipo desejado.

TÓPICO AVANÇADO 14.5

A interface `Comparator`

O Tópico Avançado 14.5 descreve a interface `Comparator`. Você gostará de utilizar um `Comparator` para classificar objetos de classes que não implementam o método `compareTo` ou que não implementam a comparação que você pretende utilizar.

RESUMO DO CAPÍTULO

1. O algoritmo de classificação por seleção classifica um array localizando repetidamente o menor elemento da região inferior não classificada e deslocando-o para frente.
2. Cientistas da computação utilizam a notação big-Oh $f(n) = O(g(n))$ para expressar que a função f não cresce mais rápido do que a função g.
3. A classificação por seleção é um algoritmo $O(n^2)$. Dobrar o tamanho do conjunto de dados significa um aumento de quatro vezes no tempo de processamento.
4. O algoritmo de classificação por intercalação classifica um array dividindo-o ao meio, classificando recursivamente cada metade e, então, intercalando as metades classificadas.
5. A classificação por intercalação é um algoritmo $O(n \log(n))$. A função $n \log(n)$ cresce muito mais lentamente do que n^2.
6. A classe `Arrays` implementa um método de classificação que você deve utilizar em seus programas Java.
7. Uma pesquisa linear examina todos os valores em um array até localizar uma correspondência ou alcançar o fim.
8. Uma pesquisa linear localiza um valor em um array em $O(n)$ passos.
9. Uma pesquisa binária localiza um valor em um array classificado determinando se o valor ocorre na primeira ou segunda metade, então repetindo a pesquisa em uma das metades.

10. Uma pesquisa binária localiza um valor em um array em $O(\log(n))$ passos.
11. O método `sort` da classe `Arrays` classifica objetos de classes que implementam a interface `Comparable`.
12. A classe `Collections` contém um método `sort` que pode classificar listas de array.

LEITURA ADICIONAL

1. Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in Java, 3rd edition,* John Wiley & Sons, 2003.*

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.lang.Comparable<T>
   compareTo
java.lang.System
   currentTimeMillis
java.util.Arrays
   binarySearch
   sort
   toString
java.util.Collections
   binarySearch
   sort
java.util.Comparator<T>
   compare
```

EXERCÍCIOS DE REVISÃO

★★ **Exercício R14.1.** *Verificando erros "por um"*. Ao escrever o algoritmo de classificação por seleção da Seção 14.1, um programador deve fazer as escolhas comuns entre `<` e `<=`, entre `a.length` e `a.length - 1` e entre `from` e `from + 1`. Esse é um solo fértil para erros *"por um"*. Faça revisões de código com arrays de comprimento 0, 1, 2 e 3, e verifique atentamente se todos os valores de índice estão corretos.

★ **Exercício R14.2.** Qual é a diferença entre pesquisa e classificação?

★★ **Exercício R14.3.** Para as seguintes expressões, qual é a ordem de crescimento de cada?

a. $n^2 + 2n + 1$
b. $n^{10} + 9n^9 + 20n^8 + 145n^7$
c. $(n + 1)^4$
d. $(n^2 + n)^2$

* N. de R.: A 4ª edição desta obra foi publicada em português pela Bookman Editora com o título *Estruturas de dados e algoritmos em Java*.

e. $n + 0.001n^3$
f. $n^3 - 1000n^2 + 10^9$
g. $n + \log(n)$
h. $n^2 + n\log(n)$
i. $2^n + n^2$
j. $\dfrac{n^3 + 2n}{n^2 + 0.75}$

★ **Exercício R14.4.** Determinamos que o número exato de visitas no algoritmo de classificação por seleção é:

$$T(n) = \frac{1}{2}n^2 + \frac{5}{2}n - 3$$

Caracterizamos esse método como apresentando crescimento $O(n^2)$. Calcule as relações:

$$T(2.000)/T(1.000)$$
$$T(4.000)/T(1.000)$$
$$T(10.000)/T(1.000)$$

e compare-as com

$$f(2.000)/f(1.000)$$
$$f(4.000)/f(1.000)$$
$$f(10.000)/f(1.000)$$

onde $f(n) = n^2$.

★ **Exercício R14.5.** Suponha que o algoritmo A leve 5 segundos para tratar um conjunto de dados de 1.000 registros. Se o algoritmo A for um algoritmo $O(n)$, quanto tempo ele levará para tratar um conjunto de dados de 2.000 registros? E um de 10.000 registros?

★★ **Exercício R14.6.** Suponha que um algoritmo leve 5 segundos para tratar um conjunto de dados de 1.000 registros. Preencha a tabela a seguir, que mostra o crescimento aproximado dos tempos de execução dependendo da complexidade do algoritmo.

	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n\log(n))$	$O(2^n)$
1.000	5	5	5	5	5
2.000					
3.000		45			
10.000					

Por exemplo, como $3.000^2/1.000^2 = 9$, o algoritmo levaria 9 vezes mais tempo, ou 45 segundos, para tratar um conjunto de dados de 3.000 registros.

★★ **Exercício R14.7.** Classifique as seguintes taxas de crescimento, do crescimento mais lento para o mais rápido.

$O(n)$	$O(n \log(n))$
$O(n^3)$	$O(2^n)$
$O(n^n)$	$O(\sqrt{n})$
$O(\log(n))$	$O(n\sqrt{n})$
$O(n^2 \log(n))$	$O(n^{\log(n)})$

★ **Exercício R14.8.** Qual é a taxa de crescimento do algoritmo padrão para localizar o valor mínimo de um array? E para localizar o valor mínimo e o máximo?

★ **Exercício R14.9.** Qual é a taxa de crescimento do seguinte método?

```
public static int count(int[] a, int c)
{
   int count = 0;
   for (int i = 0; i < a.length; i++)
   {
      if (a[i] == c) count++;
   }
   return count;
}
```

★★ **Exercício R14.10.** Sua tarefa é remover todas as duplicatas de um array. Por exemplo, se o array tiver os valores

4 7 11 4 9 5 11 7 3 5

então o array deveria ser convertido em

4 7 11 9 5 3

Eis um algoritmo simples. Veja `a[i]`. Conte quantas vezes ele ocorre em `a`. Se a contagem for maior do que 1, remova-o. Qual é a taxa de crescimento do tempo necessário para esse algoritmo?

★★ **Exercício R14.11.** Considere o seguinte algoritmo para remover todas as duplicatas de um array. Classifique o array. Para cada elemento no array, examine seu próximo vizinho para decidir se ele está presente mais de uma vez. Se estiver, remova-o. Esse algoritmo é mais rápido do que aquele do Exercício R14.10?

★★★ **Exercício R14.12.** Desenvolva um algoritmo $O(n \log(n))$ para remover duplicatas de um array se o array resultante precisar ter a mesma ordenação do array original.

★★★ **Exercício R14.13.** Por que o desempenho da classificação por inserção é significativamente melhor do que o da classificação por seleção quando um array já está classificado?

★★★ **Exercício R14.14.** Considere o seguinte aumento de velocidade do algoritmo de classificação por inserção do Tópico Avançado 14.1. Para cada elemento, utilize o algoritmo de pesquisa binária aprimorado que fornece a posição de inserção para elementos ausentes. Esse aumento de velocidade tem um impacto significativo na eficiência do algoritmo?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★ **Exercício P14.1.** Modifique o algoritmo de classificação por seleção para classificar um array de inteiros em ordem decrescente.

★ **Exercício P14.2.** Modifique o algoritmo de classificação por seleção para classificar um array de moedas por seu valor.

★★ **Exercício P14.3.** Escreva um programa que gera a tabela de tempos de execução da classificação por seleção automaticamente. O programa deve pedir o menor e o maior valor de `n` e o número de medidas e então fazer todas as execuções de exemplo.

★ **Exercício P14.4.** Modifique o algoritmo de classificação por intercalação para classificar um array de strings em ordem lexicográfica.

★★★ **Exercício P14.5.** Escreva um programa de pesquisa de telefone. Leia um conjunto de dados com 1.000 nomes e números de telefone de um arquivo que contém os números em ordem aleatória. Trate pesquisas por nome e também inverta pesquisas por número de telefone. Utilize uma pesquisa binária para ambas as pesquisas.

★★ **Exercício P14.6.** Implemente um programa que mede o desempenho do algoritmo de classificação por inserção descrito no Tópico Avançado 14.1.

★★★ **Exercício P14.7.** Escreva um programa que classifica um `ArrayList<Coin>` em ordem decrescente de modo que a moeda mais valiosa esteja no começo do array. Utilize um `Comparator`.

★★ **Exercício P14.8.** Considere o algoritmo de pesquisa binária na Seção 14.7. Se nenhuma correspondência for encontrada, o método `search` retorna –1. Modifique o método de modo que, se a não for localizado, o método retorna $-k - 1$, onde k é a posição antes da qual o elemento deveria ser inserido. (Esse é o mesmo comportamento de `Arrays.binarySearch`.)

★★ **Exercício P14.9.** Implemente o método `sort` do algoritmo de classificação por intercalação sem recursão, onde o comprimento do array é uma potência de 2. Primeiro intercale regiões adjacentes de tamanho 1, depois regiões adjacentes de tamanho 2, regiões adjacentes de tamanho 4 e assim por diante.

★★★ **Exercício P14.10.** Implemente o método `sort` do algoritmo de classificação por intercalação sem recursão, onde o comprimento do array é um número arbitrário. Continue intercalando regiões adjacentes cujas dimensões sejam uma potência de 2 e preste atenção especial à última área cujo tamanho é menor.

★★★ **Exercício P14.11.** Utilize a classificação por inserção e a pesquisa binária do Exercício P14.8 para classificar um array conforme descrito no Exercício R14.14. Implemente esse algoritmo e meça seu desempenho.

★ **Exercício P14.12.** Forneça uma classe `Person` que implementa a interface `Comparable`. Compare as pessoas usando seus nomes. Peça ao usuário para inserir 10 nomes e gere 10 objetos `Person`. Utilizando o método `compareTo`, determine a primeira e a última pessoa entre eles e as imprima.

★★ **Exercício P14.13.** Classifique uma lista de array de strings aumentando o *comprimento*. *Dica:* forneça um `Comparator`.

★★★ **Exercício P14.14.** Classifique uma lista de array de strings aumentando o comprimento, de modo que strings com o mesmo comprimento sejam classificadas lexicograficamente. *Dica:* forneça um `Comparator`.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 14.1.** Escreva um programa que mantém uma agenda. Faça uma classe `Appointment` que armazena a descrição do compromisso, o dia do compromisso, o horário de início e o de término. Seu programa deve manter os compromissos em uma lista de arrays classificada. Os usuários podem adicionar compromissos e imprimir todos os compromissos de um certo dia. Quando um novo compromisso for adicionado, use pesquisa binária para localizar onde ele deve ser inserido na lista de arrays. Não o adicione se ele entrar em conflito com outro compromisso.

★★★G **Projeto 14.2.** Implemente uma *animação gráfica* de algoritmos de classificação e pesquisa. Preencha um array com um conjunto de números aleatórios entre 1 e 100. Desenhe cada elemento do array como uma barra, como na Figura 3. Sempre que o algoritmo alterar o array, espere o usuário clicar em um botão, depois chame o método `repaint`.

Anime a classificação por seleção, a classificação por intercalação e a pesquisa binária. Na animação da pesquisa binária, destaque o elemento atualmente inspecionado e os valores atuais de `from` e `to`.

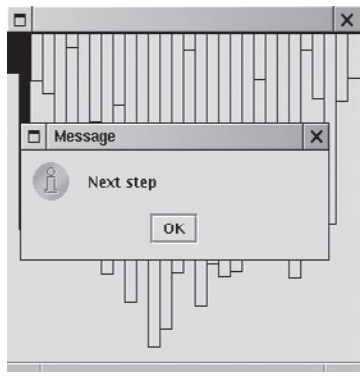


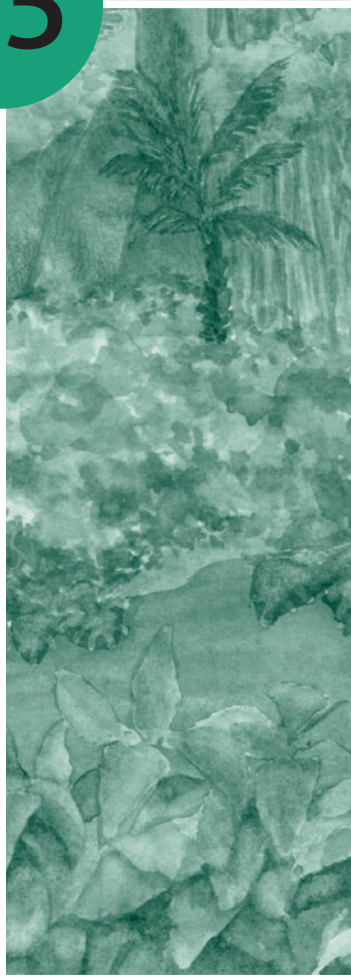
Figura 3
Animação gráfica.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Descartar a variável `temp` não funcionaria. Então `a[i]` e `a[j]` acabariam com o mesmo valor.
2. 1 | 5 4 3 2 6, 1 2 | 4 3 5 6, 1 2 3 4 5 6
3. Quatro vezes o tempo de 40.000 valores, ou aproximadamente 50 segundos.
4. Uma parábola.
5. Requer aproximadamente 100 vezes mais tempo.
6. Se n for 4, então $\frac{1}{2}n^2$ é 8 e $\frac{5}{2}n - 3$ é 7.
7. Quando o laço `while` anterior terminar, a condição de laço deve ser falsa, isto é, `iFirst >= first.length` ou `iSecond >= second.length` (Lei de De Morgan). Então `first.length - iFirst <= 0` ou `iSecond.length - iSecond <= 0`.
8. Primeiro classifique 8 7 6 5. Recursivamente, classifique primeiro 8 7. Recursivamente, primeiro 8. Ele já está classificado. Classifique 7. Ele já está classificado. Intercale-os: 7 8. Faça o mesmo com 6 5 para obter 5 6. Intercale-os em 5 6 7 8. Faça o mesmo com 4 3 2 1: classifique 4 3 classificando 4 e 3 e intercalando-os como 3 4. Classifique 2 1 classificando 2 e 1 e intercalando-os como 1 2. Intercale 3 4 e 1 2 como 1 2 3 4. Por fim, intercale 5 6 7 8 e 1 2 3 4 como 1 2 3 4 5 6 7 8.
9. Aproximadamente 100.000 · log(100.000) / 50.000 · log(50.000) = 2 · 5 / 4,7 = 2,13 vezes o tempo requerido para 50.000 valores. Isso é 2,13 · 97 milissegundos ou aproximadamente 207 milissegundos.
10. Chamando `Arrays.sort(values)`.
11. Em geral, você faria 500.000 comparações.
12. O método `search` retorna o índice em que ocorre a correspondência, não os dados armazenados nessa localização.
13. Você pesquisaria aproximadamente 20. (O log binário de 1.024 é 10.)
14. Então você sabe onde inseri-lo de modo que o array permaneça classificado e você possa continuar usando a pesquisa binária.
15. Caso contrário, você não saberia se um valor está ou não presente quando o método retornasse 0.
16. A classe `Rectangle` não implementa a interface `Comparable`.
17. A classe `BankAccount` precisa implementar a interface `Comparable`. Seu método `compareTo` deve comparar os saldos bancários.

Capítulo 15

Uma Introdução a Estruturas de Dados



Objetivos do capítulo

- Aprender a utilizar as listas encadeadas fornecidas na biblioteca padrão
- Saber utilizar iteradores para percorrer listas encadeadas
- Entender a implementação de listas encadeadas
- Distinguir entre tipos de dados abstratos e concretos
- Conhecer a eficiência de operações fundamentais sobre listas e arrays
- Familiarizar-se com os tipos pilha e fila

Até agora, utilizamos arrays como um mecanismo do tipo "um tamanho para todos" a fim de coletar objetos. Entretanto, cientistas da computação desenvolveram muitas estruturas de dados com vantagens variáveis em termos de desempenho. Neste capítulo, você aprenderá sobre a *lista encadeada*, uma estrutura de dados que permite adicionar e remover elementos de modo eficiente, sem mover nenhum elemento existente. Você também aprenderá a distinção entre tipos de dados abstratos e concretos. Um tipo abstrato define em detalhes as operações fundamentais que devem ser suportadas eficientemente, mas deixa a implementação não-especificada. Os tipos pilha e fila, introduzidos no final deste capítulo, são exemplos de tipos abstratos.

Conteúdo do Capítulo

15.1 Utilizando listas encadeadas 604

Tópico avançado 15.1: A interface `Iterable` e o laço "For Each" ⊕

15.2 Implementando listas encadeadas 609

Tópico avançado 15.2: Classes internas estáticas ⊕

15.3 Tipos abstratos de dados e tipos concretos de dados 619

15.4 Pilhas e filas 623

Fato aleatório 15.1: Padronização ⊕

15.1 Utilizando listas encadeadas

Uma *lista encadeada* é uma estrutura de dados utilizada para coletar uma seqüência de objetos, o que permite a adição e remoção eficiente de elementos no meio da seqüência.

Para entender a necessidade de tal estrutura de dados, imagine um programa que mantém uma seqüência de objetos empregado, classificados pelo sobrenome dos empregados. Quando um novo empregado é contratado, um objeto precisa ser inserido na seqüência. A menos que a empresa contrate pessoas em ordem alfabética, é provável que o novo objeto precise ser inserido em algum lugar no meio da seqüência. Se utilizarmos um array para armazenar os objetos, então todos os objetos posicionados depois do novo objeto devem ser movidos para o final.

Inversamente, se um empregado deixar a empresa, o objeto deve ser removido e o espaço vazio na seqüência precisa ser fechado, movendo todos os objetos que vêm depois dele. Mover um grande número de valores pode envolver uma quantidade substancial de tempo de processamento. Gostaríamos de estruturar os dados de maneira que minimizasse esse custo.

Uma lista encadeada consiste em um número de nós, cada um dos quais tem uma referência ao próximo nó.

É eficiente adicionar e remover elementos no meio de uma lista encadeada.

É eficiente visitar os elementos de uma lista encadeada em ordem seqüencial, mas não em acesso aleatório.

Em vez de armazenar os valores em um array, uma lista encadeada usa uma seqüência de *nós*. Cada nó armazena um valor e uma referência ao próximo nó na seqüência (veja Figura 1). Quando você insere um novo nó em uma lista encadeada, somente as referências de nó vizinhas precisam ser atualizadas. O mesmo é verdadeiro quando você remove um nó. Qual é o truque? Listas encadeadas permitem inserção e remoção rápida, mas o acesso aos elementos pode ser lento.

Por exemplo, suponha que você queira localizar o quinto elemento. Primeiro você deve percorrer os quatro primeiros. É um problema se você precisar acessar os elementos em ordem arbitrária. O termo "acesso aleatório" é utilizado na ciência da computação para descrever um padrão de acesso em que elementos são acessados em ordem arbitrária (não necessariamente aleatória). Ao contrário, o acesso seqüencial visita os elementos em seqüência. Por exemplo, uma pesquisa binária requer acesso aleatório, enquanto uma pesquisa linear requer acesso seqüencial.

Naturalmente, se na maioria das vezes você visitar todos os elementos em seqüência (por exemplo, para exibir ou imprimir os elementos), a ineficiência do acesso aleatório não é

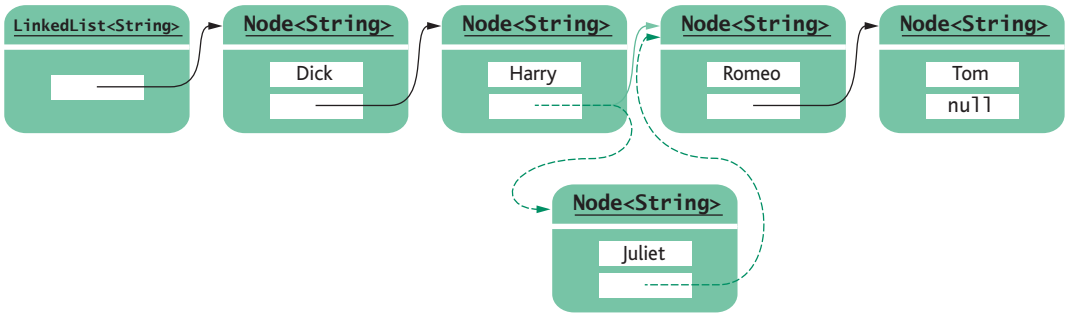


Figura 1 Inserindo um elemento em uma lista encadeada.

um problema. Você usa listas encadeadas quando está preocupado com a eficiência ao inserir ou remover elementos e raramente precisa acessar elementos em ordem aleatória.

A biblioteca Java fornece uma classe lista encadeada. Nesta seção, você aprenderá a utilizar a classe da biblioteca. Na próxima, você abrirá o capô e verá como alguns de seus métodos-chave são implementados.

A classe `LinkedList` no pacote `java.util` é uma classe genérica, exatamente como a classe `ArrayList`. Isto é, você especifica o tipo dos elementos de lista entre os sinais de menor e maior, como em `LinkedList<String>` ou `LinkedList<Product>`.

Os seguintes métodos fornecem acesso direto ao primeiro e ao último elemento da lista. Aqui, `E` é o tipo de elemento `LinkedList<E>`.

```
void addFirst(E element)
void addLast(E element)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

Como adicionar e remover elementos no meio da lista? A lista não lhe dará referências aos nós. Se tivesse acesso direto a eles e os manipulasse de alguma maneira, você quebraria a lista encadeada. Como verá na próxima seção, onde você mesmo implementa algumas operações de lista encadeada, manter intactos todos os links entre os nós não é uma tarefa trivial.

Você usa um iterador de lista para acessar elementos em uma lista encadeada.

Em vez disso, a biblioteca Java fornece um tipo `ListIterator`. Um iterador de lista encapsula uma posição em qualquer lugar dentro da lista encadeada (veja Figura 2).

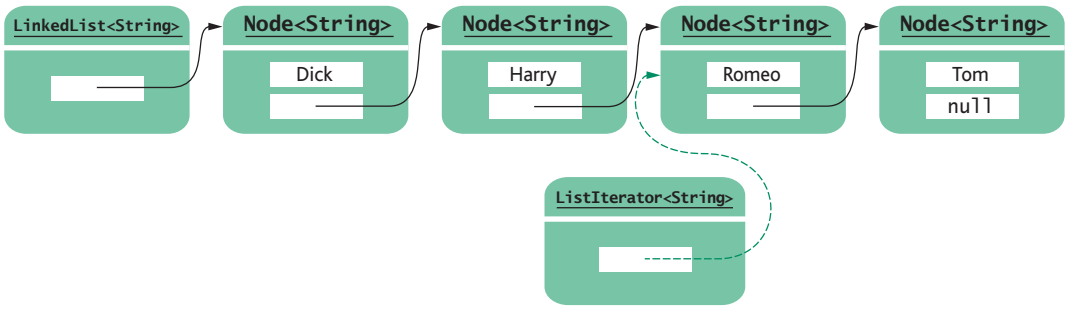


Figura 2 Um iterador de lista.

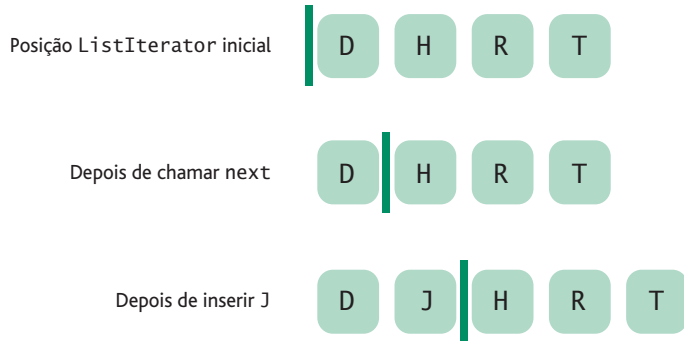


Figura 3 Visualização conceitual do iterador de lista.

Conceitualmente, você deve pensar no iterador como algo que aponta entre dois elementos, assim como o cursor em um processador de texto aponta entre dois caracteres (veja Figura 3). Na visualização conceitual, pense em cada elemento como sendo uma letra em um processador de texto e pense no iterador como sendo o cursor intermitente entre as letras.

Você obtém um iterador de lista com o método `listIterator` da classe `LinkedList`:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Observe que a classe iteradora também é um tipo genérico. Um `ListIterator<String>` itera por uma lista de strings; um `ListIterator<Product>` visita os elementos em uma `LinkedList<Product>`.

Inicialmente, o iterador aponta para antes do primeiro elemento. Você pode mover a posição do iterador com o método `next`:

```
iterator.next();
```

O método `next` lança uma `NoSuchElementException` se você já tiver passado do fim da lista. Você sempre deve chamar o método `hasNext` antes de chamar `next` – ele retorna `true` se houver um próximo elemento.

```
if (iterator.hasNext())
   iterator.next();
```

O método `next` retorna o elemento pelo qual o iterador está passando. Quando você utilizar um `ListIterator<String>`, o tipo de retorno do método `next` é `String`. Em geral, o tipo de retorno do método `next` corresponde ao parâmetro de tipo.

Você percorre todos os elementos em uma lista encadeada de strings com o seguinte laço:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   Faz algo com name
}
```

Se seu laço simplesmente visitar todos os elementos da lista encadeada, você pode utilizar o laço “for each”:

```
for (String name : employeeNames)
{
   Faz algo com name
}
```

Então você não precisa se preocupar com iteradores. Nos bastidores, o laço `for` usa um iterador para visitar todos os elementos da lista (veja o Tópico Avançado 15.1).

Os nós da classe `LinkedList` armazenam dois links: um para o próximo elemento e o outro para o elemento anterior. Tal lista é chamada de lista duplamente encadeada. Você pode utilizar os métodos `previous` e `hasPrevious` da interface `ListIterator` para fazer o iterador andar para trás.

O método `add` adiciona um objeto depois do iterador e, logo após, move a posição do iterador além do novo elemento.

```
iterator.add("Juliet");
```

Você pode visualizar a inserção como a digitação de um texto em um processador de texto. Cada caractere é inserido depois do cursor e, então, o cursor passa para a frente do caractere inserido (veja Figura 3). A maioria das pessoas nunca presta muita atenção nisso – é recomendável que você experimente e observe atentamente como seu processador de texto insere caracteres.

O método `remove` remove o objeto que foi retornado pela última chamada a `next` ou `previous`. Por exemplo, o seguinte laço remove todos os nomes que satisfazem certa condição:

```
while (iterator.hasNext())
{
   String name = iterator.next();
   if (name satisfaz a condição)
      iterator.remove();
}
```

Você precisa ter cuidado ao chamar `remove`. Ele só pode ser chamado uma vez depois de chamar `next` ou `previous` e você não pode chamá-lo imediatamente depois de uma chamada a `add`. Se você chamar o método indevidamente, ele lança uma `IllegalStateException`.

Eis um programa de exemplo que insere strings em uma lista e, então, itera pela lista, adicionando e removendo elementos. Por fim, a lista inteira é impressa. Os comentários indicam a posição de iterador.

ch15/uselist/ListTester.java

```
1  import java.util.LinkedList;
2  import java.util.ListIterator;
3
4  /**
5     Um programa que testa a classe LinkedList.
6  */
7  public class ListTester
8  {
9     public static void main(String[] args)
10    {
11       LinkedList<String> staff = new LinkedList<String>();
12       staff.addLast("Dick");
13       staff.addLast("Harry");
```

```
14         staff.addLast("Romeo");
15         staff.addLast("Tom");
16
17         // | nos comentários indica a posição do iterador
18
19         ListIterator<String> iterator
20               = staff.listIterator(); // |DHRT
21         iterator.next(); // D|HRT
22         iterator.next(); // DH|RT
23
24         // Adiciona mais elementos depois do segundo elemento
25
26         iterator.add("Juliet"); // DHJ|RT
27         iterator.add("Nina"); // DHJN|RT
28
29         iterator.next(); // DHJNR|T
30
31         // Remove o último elemento visitado
32
33         iterator.remove(); // DHJN|T
34
35         // Imprime todos os elementos
36
37         for (String name : staff)
38            System.out.print(iterator.next() + " ");
39         System.out.println();
40         System.out.println("Expected: Dick Harry Juliet Nina Tom");
41      }
42   }
```

Saída

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Autoverificação da aprendizagem

1. As listas encadeadas ocupam mais espaço de armazenamento do que arrays do mesmo tamanho?
2. Por que não precisamos de iteradores com arrays?

Tópico avançado 15.1

A interface `Iterable` e o laço "for each"

O Tópico Avançado 15.1 discute a interface `Iterable`. O laço "for each" pode ser aplicado a qualquer objeto que implementa a interface `Iterable`.

15.2 Implementando listas encadeadas

Na última seção você viu como utilizar a classe lista encadeada fornecida pela biblioteca Java. Nesta seção, veremos a implementação de uma versão simplificada dessa classe. Isso mostra como as operações de lista manipulam os links quando a lista é modificada.

Para manter esse código de exemplo simples, não implementaremos todos os métodos da classe lista encadeada. Implementaremos somente uma lista simplesmente encadeada, e a classe lista fornecerá acesso direto somente ao primeiro elemento da lista, não ao último. Nossa lista não utilizará um parâmetro de tipo. Iremos simplesmente armazenar valores `Object` brutos e usar coerções ao recuperá-los. O resultado será uma classe lista completamente funcional que mostra como os links são atualizadas nas operações `add` e `remove` e como o iterador percorre a lista.

Um objeto `Node` armazena um objeto e uma referência ao próximo nó. Como os métodos, tanto da classe lista encadeada como da classe iteradora, têm acesso freqüente às variáveis de instância `Node`, não tornaremos as variáveis de instância privadas. Em vez disso, faremos uma classe interna privada `Node` da classe `LinkedList`. Como nenhum dos métodos de lista retorna um objeto `Node`, é seguro deixar as variáveis de instância públicas.

```
public class LinkedList
{
   . . .
   private class Node
   {
      public Object data;
      public Node next;
   }
}
```

A classe `LinkedList` armazena uma referência `first` ao primeiro nó (ou `null`, se a lista estiver completamente vazia).

```
public class LinkedList
{
   public LinkedList()
   {
      first = null;
   }
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }
   . . .
   private Node first;
}
```

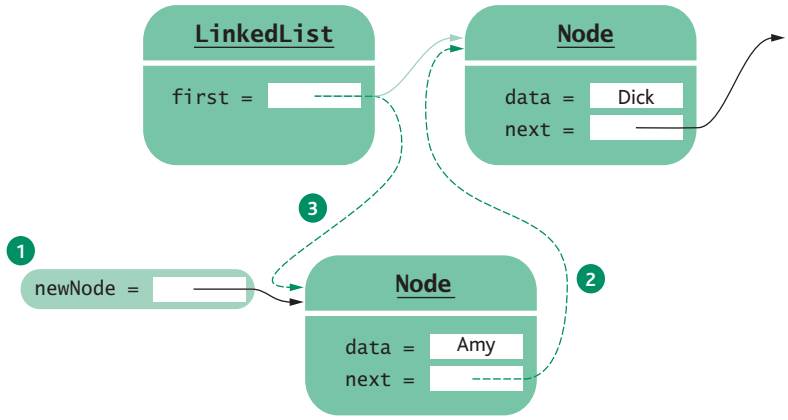


Figura 4 Adicionando um nó no começo de uma lista encadeada.

Agora vamos nos voltar para o método `addFirst` (veja Figura 4). Quando um novo nó é adicionado à lista, ele se torna o começo da lista e o nó que era o antigo começo da lista, torna-se seu próximo nó:

```
public class LinkedList
{
   . . .
   public void addFirst(Object element)
   {
     Node newNode = new Node(); 1
     newNode.data = element;
     newNode.next = first; 2
     first = newNode; 3
   }
   . . .
}
```

Remover o primeiro elemento da lista funciona da seguinte maneira. Os dados do primeiro nó são salvos e mais tarde são retornados como o resultado do método. O sucessor do primeiro nó torna-se o primeiro nó da lista mais curta (veja Figura 5). Então não há mais referências ao antigo nó, e o coletor de lixo irá, por fim, reciclá-lo.

```
public class LinkedList
{
   . . .
   public Object removeFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      Object element = first.data;
      first = first.next; 1
      return element;
   }
   . . .
}
```

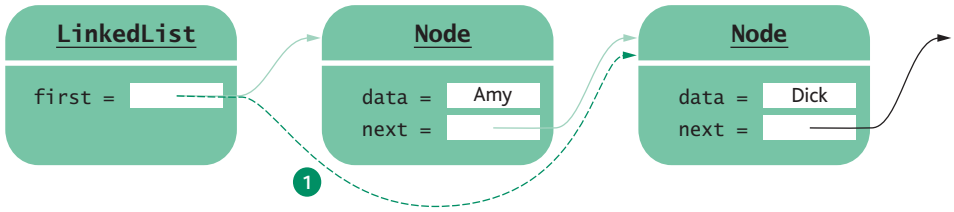


Figura 5 Removendo o primeiro nó de uma lista encadeada.

Agora, voltemos para a classe iteradora. A interface `ListIterator` na biblioteca padrão define nove métodos. Omitimos quatro deles (os métodos que retrocedem o iterador e os métodos que informam um índice de inteiro do iterador).

Nossa classe `LinkedList` define uma classe `LinkedListIterator` interna e privada, que implementa a interface `ListIterator` simplificada. Como `LinkedListIterator` é uma classe interna, ela tem acesso aos recursos privados da classe `LinkedList` – em particular, o campo `first` e a classe `Node` privada.

Observe que os clientes da classe `LinkedList`, de fato, não sabem o nome da classe iteradora. Eles só sabem que é uma classe que implementa a interface `ListIterator`.

```
public class LinkedList
{
   . . .
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }

   private class LinkedListIterator
      implements ListIterator
   {
      public LinkedListIterator()
      {
         position = null;
         previous = null;
      }

      . . .
      private Node position;
      private Node previous;
   }
   . . .
}
```

Cada objeto iterador tem uma referência `position` ao último nó visitado. Também armazenamos uma referência ao último nó antes desse. Precisaremos dessa referência para ajustar os links adequadamente no método `remove`.

O método `next` é simples. A referência `position` é avançada para `position.next` e a antiga posição é lembrada em `previous`. Porém, há um caso especial – se o iterador apontar antes do primeiro elemento da lista, então a antiga `position` é `null` e `position` deve ser configurada como `first`.

```
private class LinkedListIterator
   implements ListIterator
{
   . . .
   public Object next()
   {
      if (!hasNext())
         throw new NoSuchElementException();
      previous = position; // Lembrete para remove

      if (position == null)
         position = first;
      else
         position = position.next;

      return position.data;
   }
   . . .
}
```

O método `next` deve ser chamado apenas quando o iterador ainda não estiver no fim da lista. O iterador está no fim se a lista estiver vazia (isto é, `first == null`) ou se não houver nenhum elemento depois da posição atual (`position.next == null`).

```
private class LinkedListIterator
   implements ListIterator
{
   . . .
   public boolean hasNext()
   {
      if (position == null)
         return first != null;
      else
         return position.next != null;
   }
   . . .
}
```

Implementar operações que modificam uma lista encadeada é um desafio – você precisa se certificar de que esteja atualizando todas as referências a nós corretamente.

Remover o último nó visitado é mais complexo. Se o elemento a ser removido for o primeiro elemento, chamamos apenas `removeFirst`. Caso contrário, um elemento no meio da lista deve ser removido e o nó precedente precisa ter sua referência `next` atualizada para pular o elemento removido (veja Figura 6). Se a referência `previous` for igual a `position`, então essa chamada para `remove` não segue imediatamente uma chamada para `next` e lançamos uma `IllegalStateException`.

De acordo com a definição do método `remove`, é inválido chamar `remove` duas vezes na linha. Portanto, o método `remove` configura a referência `previous` como `position`.

```
private class LinkedListIterator
   implements ListIterator
{
   . . .
```

```
      public void remove()
      {
         if (previous == position)
            throw new IllegalStateException();
         if (position == first)
         {
            removeFirst();
         }
         else
         {
            previous.next = position.next; ①
         }
         position = previous; ②
      }
      . . .
   }
```

O método `set` altera o armazenamento de dados no elemento previamente visitado. Sua implementação é simples e direta porque nossas listas encadeadas podem ser percorridas apenas em uma direção. A implementação de lista encadeada da biblioteca padrão deve monitorar se o último movimento do iterador foi para frente ou para trás. Por essa razão, a biblioteca padrão proíbe uma chamada para o método `set` seguindo um método `add` ou `remove`. Nós não impomos essa restrição.

```
   public void set(Object element)
   {
      if (position == null)
         throw new NoSuchElementException();
      position.data = element;
   }
```

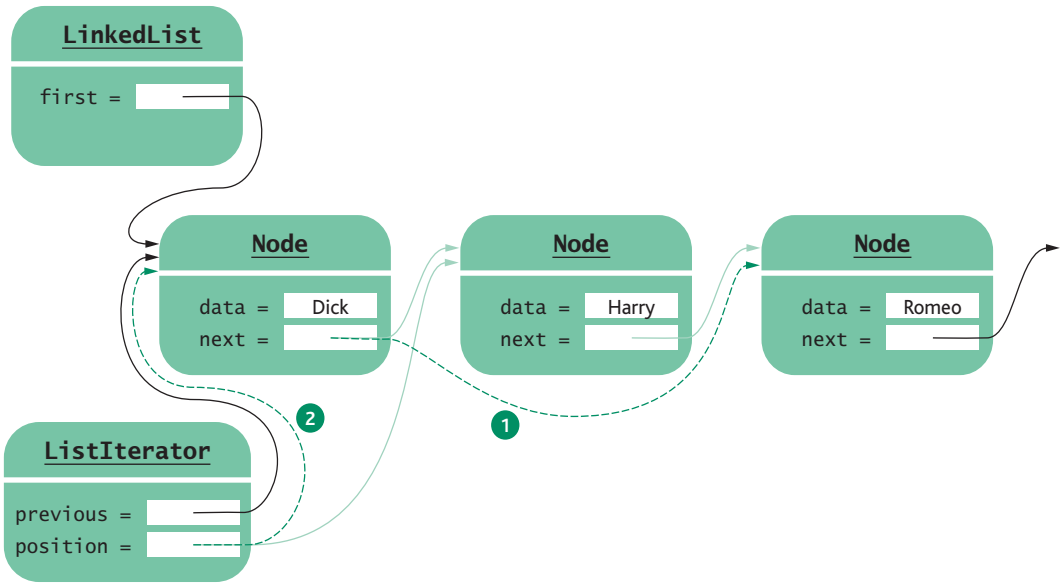


Figura 6 Removendo um nó do meio de uma lista encadeada.

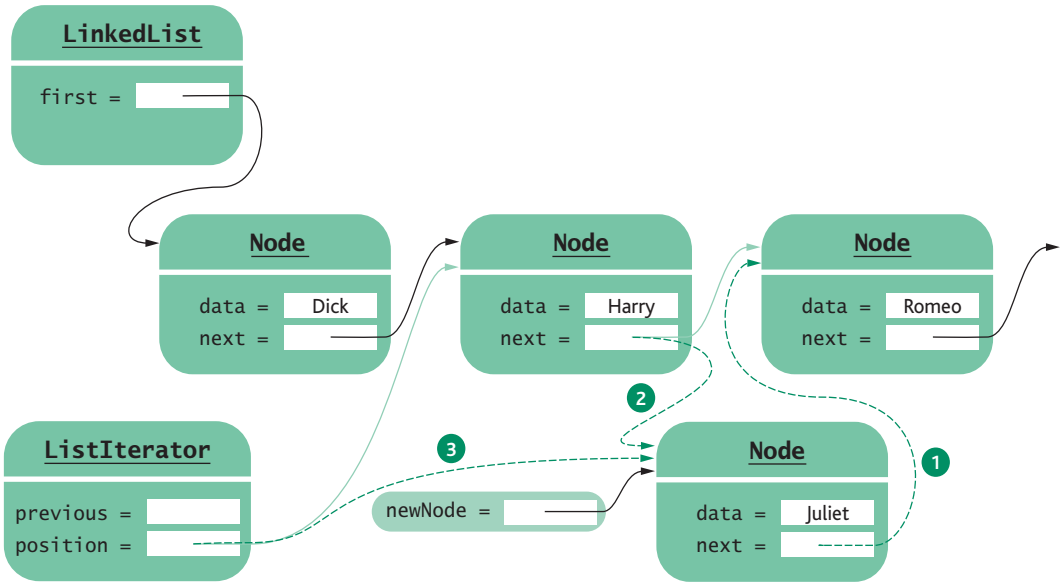


Figura 7 Adicionando um nó no meio de uma lista encadeada.

Por fim, a operação mais complexa é a adição de um nó. Você insere o novo nó depois da posição atual e configura o sucessor do novo nó como sucessor da posição atual (veja Figura 7).

```
private class LinkedListIterator
   implements ListIterator
{
   . . .
   public void add(Object element)
   {
      if (position == null)
      {
         addFirst(element);
         position = first;
      }
      else
      {
         Node newNode = new Node();
         newNode.data = element;
         newNode.next = position.next; 1
         position.next = newNode; 2
         position = newNode; 3
      }
      previous = position;
   }
   . . .
}
```

A implementação completa de nossa classe `LinkedList` está no final desta seção.

Agora você sabe utilizar a classe `LinkedList` da biblioteca Java e teve uma rápida introdução para entender como as listas encadeadas são implementadas.

ch15/impllist/LinkedList.java

```
import java.util.NoSuchElementException;

/**
   Uma lista encadeada é uma seqüência de nós com
   inserção e remoção eficiente de elementos. Essa classe
   contém um subconjunto dos métodos da classe
   java.util.LinkedList padrão.
*/
public class LinkedList
{
   /**
      Constrói uma lista encadeada vazia.
   */
   public LinkedList()
   {
      first = null;
   }

   /**
      Retorna o primeiro elemento na lista encadeada.
      @return primeiro elemento na lista encadeada
   */
   public Object getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }

   /**
      Remove o primeiro elemento da lista encadeada.
      @return elemento removido
   */
   public Object removeFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      Object element = first.data;
      first = first.next;
      return element;
   }

   /**
      Adiciona um elemento na frente da lista encadeada.
      @param element elemento a adicionar
   */
   public void addFirst(Object element)
   {
```

```
49          Node newNode = new Node();
50          newNode.data = element;
51          newNode.next = first;
52          first = newNode;
53       }
54
55       /**
56          Retorna um iterador para iterar por essa lista.
57          @return iterador para iterar por essa lista
58       */
59       public ListIterator listIterator()
60       {
61          return new LinkedListIterator();
62       }
63
64       private Node first;
65
66       private class Node
67       {
68          public Object data;
69          public Node next;
70       }
71
72       private class LinkedListIterator implements ListIterator
73       {
74          /**
75             Constrói um iterador que aponta para frente
76             da lista encadeada.
77          */
78          public LinkedListIterator()
79          {
80             position = null;
81             previous = null;
82          }
83
84          /**
85             Move o iterador além do próximo elemento.
86             @return elemento visitado
87          */
88          public Object next()
89          {
90             if (!hasNext())
91                throw new NoSuchElementException();
92             previous = position; // Lembrete para remover
93
94             if (position == null)
95                position = first;
96             else
97                position = position.next;
98
99             return position.data;
100         }
101
```

```
102         /**
103            Testa se há um elemento depois da
104            posição do iterador.
105            @return true se houver um elemento depois da
106            posição do iterador.
107         */
108         public boolean hasNext()
109         {
110            if (position == null)
111               return first != null;
112            else
113               return position.next != null;
114         }
115
116         /**
117            Adiciona um elemento antes da posição do iterador
118            e move o iterador além do elemento inserido.
119            @param element elemento a adicionar
120         */
121         public void add(Object element)
122         {
123            if (position == null)
124            {
125               addFirst(element);
126               position = first;
127            }
128            else
129            {
130               Node newNode = new Node();
131               newNode.data = element;
132               newNode.next = position.next;
133               position.next = newNode;
134               position = newNode;
135            }
136            previous = position;
137         }
138
139         /**
140            Remove o último elemento visitado. Esse método só pode
141            ser chamado depois de uma chamada ao método next()
142         */
143         public void remove()
144         {
145            if (previous == position)
146               throw new IllegalStateException();
147
148            if (position == first)
149            {
150               removeFirst();
151            }
```

```
152         else
153         {
154            previous.next = position.next;
155         }
156         position = previous;
157      }
158
159      /**
160         Configura o último elemento visitado com um
161         valor diferente.
162         @param element elemento a configurar
163      */
164      public void set(Object element)
165      {
166         if (position == null)
167            throw new NoSuchElementException();
168         position.data = element;
169      }
170
171      private Node position;
172      private Node previous;
173   }
174 }
```

ch15/impllist/ListIterator.java

```
1  /**
2     Um iterador de lista permite acesso a uma posição em uma lista encadeada.
3     Esta interface contém um subconjunto dos métodos da
4     interface java.util.ListIterator padrão. Os métodos para
5     percorrer de trás para frente não estão incluídos.
6  */
7  public interface ListIterator
8  {
9     /**
10       Move o iterador além do próximo elemento.
11       @return elemento visitado
12    */
13    Object next();
14
15    /**
16       Testa se há um elemento depois da posição do
17       iterador.
18       @return true se houver um elemento depois da
19       posição do iterador
20    */
21    boolean hasNext();
22
23    /**
24       Adiciona um elemento antes da posição do iterador
25       e move o iterador além do elemento inserido.
26       @param element elemento a adicionar
27    */
28    void add(Object element);
```

```
29
30      /**
31          Remove o último elemento visitado. Este método só pode
32          ser chamado depois de uma chamada ao método next().
33      */
34      void remove();
35
36      /**
37          Configura o último elemento visitado com um diferente
38          valor.
39          @param element o elemento a configurar
40      */
41      void set(Object element);
41  }
```

AUTOVERIFICAÇÃO DA APRENDIZAGEM

3. Rastreie o método `addFirst` ao adicionar um elemento a uma lista vazia.
4. Conceitualmente, um iterador aponta entre elementos (veja Figura 3). A referência de posição aponta para o elemento à esquerda ou à direita?
5. Por que o método `add` tem dois casos separados?

TÓPICO AVANÇADO 15.2

Classes internas estáticas

O Tópico Avançado 15.2 mostra como você pode fazer a classe interna `Node` um pouco mais eficiente declarando-a como uma classe interna estática.

15.3 Tipos abstratos de dados e tipos concretos de dados

Há duas maneiras de ver uma lista encadeada. Uma maneira é pensar na implementação concreta de tal lista como uma seqüência de objetos do tipo do nó com links entre eles (veja Figura 8).

Um tipo abstrato de dados define as operações fundamentais nos dados, mas não especifica uma implementação.

Por outro lado, você pode considerar o conceito *abstrato* da lista encadeada. No tipo abstrato, uma lista encadeada é uma seqüência ordenada de itens de dados que pode ser percorrida com um iterador (veja Figura 9).

Da mesma forma, há duas maneiras de ver uma lista de arrays. Naturalmente, uma lista de arrays tem uma implementação concreta: um array parcialmente preenchido de referências a objetos (veja Figura 10). Mas, em geral, você não pensa na implementação concreta ao usar uma lista de arrays.

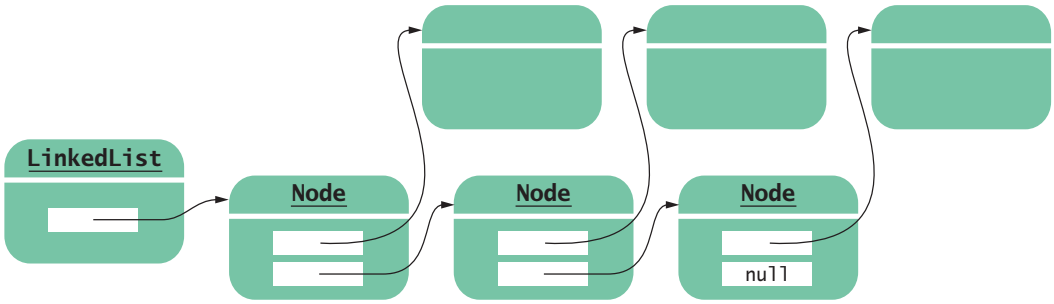


Figura 8 Uma visão concreta de uma lista encadeada.

Figura 9 Uma visão abstrata de uma lista encadeada.

Você adota o ponto de vista abstrato. Uma lista de arrays é uma seqüência ordenada de itens de dados, cada um dos quais podendo ser acessado por um índice de inteiros (veja Figura 11).

As implementações concretas de uma lista encadeada e uma lista de arrays são bem diferentes. As abstrações, por outro lado, parecem semelhantes à primeira vista. Para ver a diferença, considere as interfaces públicas reduzidas aos seus elementos essenciais.

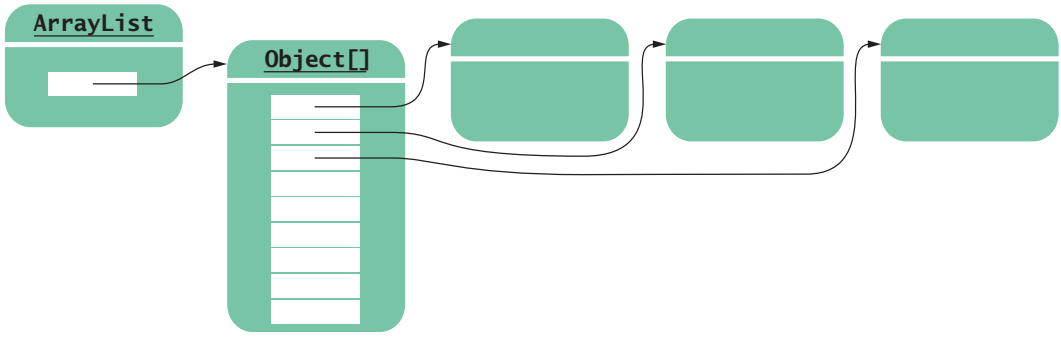


Figura 10 Uma visão concreta de uma lista de arrays.

Figura 11 Uma visão abstrata de uma lista de arrays.

Uma lista de arrays permite *acesso aleatório* a todos os elementos. Você especifica um índice do tipo inteiro e pode obter (*get*) ou configurar (*set*) o elemento correspondente.

```
public class ArrayList
{
   public Object get(int index) { . . . }
   public void set(int index, Object element) { . . . }
   . . .
}
```

Com uma lista encadeada, por outro lado, o acesso a elementos é um pouco mais complexo. Uma lista encadeada permite acesso seqüencial. Você precisa solicitar à lista encadeada um iterador. Utilizando esse iterador, você pode percorrer facilmente os elementos da lista individualmente. Mas, se quiser ir para um determinado elemento, digamos o centésimo, primeiro você precisa pular todos os elementos antes dele.

```
public class LinkedList
{
   public ListIterator listIterator() { . . . }
   . . .
}

public interface ListIterator
{
   Object next();
   boolean hasNext();
   void add(Object element);
   void remove();
   void set(Object element);
   . . .
}
```

Aqui nós só mostramos as operações *fundamentais* nas listas de arrays e listas encadeadas. É possível compor outras operações a partir dessas operações fundamentais. Por exemplo, adicionar ou remover um elemento em uma lista de arrays movendo todos os elementos além do índice de inserção ou remoção, chamando `get` e `set` várias vezes.

Naturalmente, a classe `ArrayList` tem métodos para adicionar e remover elementos no meio, ainda que sejam lentos. Inversamente, a classe `LinkedList` tem os métodos `get` e `set` que permitem acessar qualquer elemento na lista encadeada, embora de modo muito ineficiente, realizando repetidos acessos seqüenciais.

De fato, o termo `ArrayList` significa que seus implementadores pretendiam combinar as interfaces de um array e de uma lista. De maneira meio confusa, as classes `ArrayList` e `LinkedList` implementam uma interface chamada `List` que define operações para acesso tanto aleatório como seqüencial.

Essa terminologia não está em uso comum fora da biblioteca Java. Em vez disso, vamos adotar uma terminologia mais tradicional. Chamaremos os tipos abstratos de *array* e *lista*. A biblioteca Java fornece implementações concretas `ArrayList` e `LinkedList` para esses tipos abstratos. Outras implementações concretas são possíveis em outras bibliotecas.

De fato, os arrays Java são outra implementação do tipo abstrato array.

Para entender de modo completo um tipo abstrato de dados, você precisa conhecer não só suas operações fundamentais mas também sua eficiência relativa.

Tabela 1 Eficiência de operações sobre arrays e listas

Operação	Array	Lista
Acesso aleatório	$O(1)$	$O(n)$
Percurso linear	$O(1)$	$O(1)$
Adiciona/remove um elemento	$O(n)$	$O(1)$

Uma lista abstrata é uma seqüência ordenada de itens que pode ser percorrida seqüencialmente e que permite inserção e remoção de elementos em qualquer posição.

Um array abstrato é uma seqüência ordenada de itens com acesso aleatório via um índice do tipo inteiro.

Em uma lista encadeada, um elemento pode ser adicionado ou removido em tempo constante (assumindo que o iterador já está na posição certa). Um número fixo de referências de nó precisa ser modificado para adicionar ou remover um nó, independentemente do tamanho da lista. Utilizando a notação big-Oh, uma operação que requer uma quantidade definida de tempo, independentemente do número total de elementos na estrutura, é indicada como $O(1)$. O acesso aleatório em uma lista de arrays também possui tempo $O(1)$.

Adicionar ou remover um elemento arbitrário em um array exige tempo $O(n)$, onde n é o tamanho da lista de arrays, porque em média $n/2$ elementos precisam ser movidos. O acesso aleatório a uma lista encadeada leva um tempo $O(n)$ porque em média $n/2$ elementos precisam ser pulados.

A Tabela 1 mostra essas informações para arrays e listas.

Por que considerar tipos abstratos? Ao implementar um algoritmo, você pode informar quais operações você precisa executar nas estruturas de dados que seu algoritmo manipula. Você pode então determinar o tipo abstrato que suporta essas operações eficientemente, sem se preocupar com os detalhes da implementação.

Por exemplo, suponha que você tenha uma coleção classificada de itens e queira localizar esses itens utilizando o algoritmo de pesquisa binária (veja Seção 14.7). Esse algoritmo faz um acesso aleatório no meio da coleção, seguido por outros acessos aleatórios. Portanto, o acesso aleatório rápido é essencial para o algoritmo funcionar corretamente. Assim que você descobrir que um array suporta acesso aleatório rápido e uma lista encadeada não, você procurará implementações concretas do tipo abstrato para arrays. Você não será levado a utilizar uma `LinkedList`, ainda que a classe `LinkedList` realmente forneça os métodos `get` e `set`.

Na próxima seção, você verá mais exemplos de tipos abstratos de dados.

Autoverificação da aprendizagem

6. Qual é a vantagem de visualizar um tipo abstratamente?
7. Como você esboçaria uma visão abstrata de uma lista duplamente encadeada? E uma visualização concreta?
8. O quão mais lento é o algoritmo de pesquisa binária em uma lista encadeada em comparação com o algoritmo de pesquisa linear?

15.4 Pilhas e filas

Nesta seção, consideraremos dois tipos abstratos de dados comuns que permitem inserção e remoção de itens apenas nas extremidades, não no meio. Uma pilha permite inserir e remover elementos apenas em uma extremidade, tradicionalmente chamada *topo* da pilha. Para dar uma noção de uma pilha, imagine uma pilha de livros (veja Figura 12).

Uma pilha é uma coleção de itens que são recuperados seguindo a regra "último a entrar, primeiro a sair".

Uma fila é uma coleção de itens que são recuperados seguindo a regra "primeiro a entrar, primeiro a sair".

Novos itens podem ser adicionados à parte superior da pilha. E a remoção dos itens também ocorre a partir do topo da fila. Portanto, eles são removidos na ordem oposta daquela em que eles foram adicionados, chamada de ordem *último a entrar, primeiro a sair*, ou *LIFO* (last in, first out). Por exemplo, se adicionar os itens `A`, `B` e `C` e, então, removê-los, você obterá `C`, `B` e `A`. Tradicionalmente, as operações de adição e remoção são chamadas `push` e `pop`.

Uma fila é semelhante a uma pilha, exceto que você adiciona itens a uma extremidade da fila (o final da fila) e os remove da outra (o início da fila). Para você ter uma noção, simplesmente pense numa fila real de pessoas (veja Figura 13). As pessoas entram no final da fila e esperam até chegar ao início da fila. Filas armazenam itens à moda *primeiro a entrar, primeiro a sair,* ou *FIFO* (*first in, first out*). Os itens são removidos na mesma ordem em que foram adicionados.

Há muitas utilizações para filas e pilhas em ciência da computação. O sistema Java de interface gráfica com o usuário mantém uma fila de eventos com todos os eventos, como os de mouse e teclado. Os eventos são inseridos na fila sempre que o sistema operacional notifica a aplicação sobre o evento. Outra *thread* de controle os remove da fila e os passa para os ouvintes de evento apropriados. Outro exemplo é uma fila de impressão. Uma impressora pode ser acessada por vários aplicativos, talvez executando em computadores diferentes. Se cada um dos aplicativos tentasse acessar a impressora ao mesmo tempo, esta imprimiria lixo. Em vez disso, cada aplicativo coloca todos os bytes que precisam ser enviados à impressora em um arquivo e insere esse arquivo na fila de impressão. Quando a impressora termina de imprimir um arquivo, ela recupera o próximo da fila. Portanto, os trabalhos de impressão são feitos utilizando a regra "primeiro a entrar, primeiro a sair", que é um arranjo justo para usuários de impressora compartilhada.

Figura 12 Uma pilha de livros.

Figura 13 Uma fila.

Pilhas são utilizadas quando uma regra de "último a entrar, primeiro a sair" é necessária. Por exemplo, considere um algoritmo que tenta encontrar um caminho em um labirinto. Ao encontrar um cruzamento, o algoritmo insere sua localização na pilha e explora o primeiro desvio. Se esse desvio for um beco sem saída, o algoritmo retorna sua localização ao topo da pilha. Se todos os desvios forem becos sem saída, o algoritmo remove a localização da pilha, revelando um cruzamento previamente encontrado. Outro exemplo importante é a pilha de tempo de execução que um processador ou máquina virtual mantém para organizar as variáveis de métodos aninhados. Sempre que um novo método é chamado, seus parâmetros e variáveis locais são inseridos em uma pilha. Quando o método termina, eles são removidos novamente. Essa pilha torna as chamadas de método recursivas possíveis.

Há uma classe `Stack` na biblioteca Java que implementa o tipo abstrato pilha e as operações `push` e `pop`. O seguinte código de exemplo mostra como utilizar essa classe.

```
Stack<String> s = new Stack<String>();
s.push("A");
s.push("B");
s.push("C");
// O laço a seguir imprime C, B e A
while (s.size() > 0)
   System.out.println(s.pop());
```

A classe `Stack` da biblioteca Java usa um array para implementar uma pilha. O Exercício P15.11 mostra como utilizar uma lista encadeada.

As implementações de uma fila na biblioteca padrão são projetadas para utilização com programas multiencadeados. Entretanto, é simples você mesmo implementar uma fila básica:

```
public class LinkedListQueue
{
   /**
      Constrói uma fila vazia que usa uma lista encadeada.
   */
   public LinkedListQueue()
   {
      list = new LinkedList();
   }
   /**
      Adiciona um elemento ao fim da fila.
      @param element elemento a ser adicionado
   */
   public void add(Object element)
   {
      list.addLast(element);
   }
   /**
      Remove um elemento do começo da fila.
      @return elemento removido
   */
   public Object remove()
   {
      return list.removeFirst();
   }
   /**
      Obtém o número de elementos na fila.
      @return tamanho
   */
   int size()
   {
      return list.size();
   }
   private LinkedList list;
}
```

Certamente você não vai querer utilizar um `ArrayList` para implementar uma fila. Remover o primeiro elemento de uma lista de arrays é ineficiente – todos os outros elementos devem ser movidos para frente. Entretanto, o Exercício P15.12 mostra como implementar uma fila como um array "circular" de modo eficiente, na qual todos os elementos permanecem na posição em que foram inseridos, mas os valores de índice, que indicam o começo e o fim da fila mudam quando os elementos são adicionados e removidos.

Neste capítulo, você viu os dois tipos abstratos de dados mais fundamentais, arrays e listas, e suas implementações concretas. Você também aprendeu sobre os tipos pilha e fila. No próximo capítulo, você verá os tipos de dados adicionais que requerem técnicas de implementação mais sofisticadas.

Autoverificação da aprendizagem

9. Faça um esboço do tipo abstrato fila, semelhante ao das Figuras 9 e 11.
10. Por que uma pilha não serviria para gerenciar trabalhos de impressão?

Fato aleatório 15.1

Padronização

O Fato Aleatório 15.1 discute os benefícios da padronização na ciência da computação e diz como os padrões são criados.

RESUMO DO CAPÍTULO

1. Uma lista encadeada consiste em uma quantidade de nós, cada um dos quais com uma referência ao próximo nó.
2. É eficiente adicionar e remover elementos no meio de uma lista encadeada.
3. É eficiente visitar os elementos de uma lista encadeada em ordem seqüencial, mas não em acesso aleatório.
4. Você usa um iterador de lista para acessar elementos em uma lista encadeada.
5. Implementar operações que modificam uma lista encadeada é um desafio – você precisa se certificar de que atualiza todas as referências a nó corretamente.
6. Um tipo abstrato de dados define as operações fundamentais sobre os dados, mas não especifica uma implementação.
7. Uma lista abstrata é uma seqüência ordenada de itens que pode ser percorrida seqüencialmente e que permite inserção e remoção de elementos em qualquer posição.
8. Um array abstrato é uma seqüência ordenada de itens com acesso aleatório via um índice do tipo inteiro.
9. Uma pilha é uma coleção de itens que são recuperados seguindo a regra “último a entrar, primeiro a sair”.
10. Uma fila é uma coleção de itens que são recuperados seguindo a regra “primeiro a entrar, primeiro sair”.

CLASSES, OBJETOS E MÉTODOS INTRODUZIDOS NESTE CAPÍTULO

```
java.util.Collection<E>
   add
   contains
   iterator
   remove
   size
java.util.Iterator<E>
   hasNext
   next
   remove
```

```
java.util.LinkedList<E>
   addFirst
   addLast
   getFirst
   getLast
   removeFirst
   removeLast
java.util.List<E>
   listIterator
java.util.ListIterator<E>
   add
   hasPrevious
   previous
   set
```

EXERCÍCIOS DE REVISÃO

★ **Exercício R15.1.** Explique o que o seguinte código imprime. Desenhe figuras de uma lista encadeada depois de cada passo. Apenas desenhe os links para frente, como na Figura 1.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Dick");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

★ **Exercício R15.2.** Explique o que o seguinte código imprime. Desenhe figuras de uma lista encadeada depois de cada passo. Apenas desenhe os links para frente, como na Figura 1.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addFirst("Dick");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

★ **Exercício R15.3.** Explique o que o seguinte código imprime. Desenhe figuras de uma lista encadeada depois de cada passo. Apenas desenhe os links para frente, como na Figura 1.

```
LinkedList<String> staff = new LinkedList<String>();
staff.addFirst("Harry");
staff.addLast("Dick");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

★ **Exercício R15.4.** Explique o que o seguinte código imprime. Desenhe figuras de uma lista encadeada e a posição do iterador depois de cada passo.

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Dick");
iterator.add("Harry");
iterator = staff.listIterator();
if (iterator.next().equals("Tom"))
   iterator.remove();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

★ **Exercício R15.5.** Explique o que o seguinte código imprime. Desenhe figuras de uma lista encadeada e a posição do iterador depois de cada passo.

```
LinkedList<String> staff = new LinkedList<String>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Dick");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

★★ **Exercício R15.6.** A classe lista encadeada da biblioteca Java suporta operações `addLast` e `removeLast`. Para executar essas operações eficientemente, a classe `LinkedList` tem uma referência `last` ao último nó da lista encadeada. Desenhe um diagrama "antes/depois" com as alterações nos links de uma lista encadeada sob o efeito dos métodos `addLast` e `removeLast`.

★★ **Exercício R15.7.** A classe de lista encadeada na biblioteca Java suporta iteradores bidirecionais. Para retroceder eficientemente, cada `Node` tem uma referência adicionada, `previous`, ao nó anterior na lista encadeada. Desenhe um diagrama "antes/depois" com alterações nos links de uma lista encadeada sob o efeito dos métodos `addFirst` e `removeFirst` que mostra como os links `previous` devem ser atualizados.

★★ **Exercício R15.8.** Que vantagens as listas têm sobre os arrays? Quais são suas desvantagens?

★★ **Exercício R15.9.** Suponha que você precisasse organizar uma lista telefônica para uma divisão de uma empresa. Há atualmente cerca de 6.000 empregados e você sabe que a central telefônica pode tratar mais de 10.000 números de telefone. Você espera centenas de consultas aos seus dados todos os dias. Você utilizaria um array ou uma lista para armazenar as informações?

★★ **Exercício R15.10.** Suponha que você precisasse manter uma série de compromissos. Você utilizaria uma lista ou um array de objetos `Appointment`?

★ **Exercício R15.11.** Suponha que você escreva um programa que modela um baralho de cartas. As cartas são tiradas da parte superior do baralho e distribuídas para os jogadores. Quando as cartas são retornadas ao baralho, elas são colocadas na parte inferior. Você armazenaria as cartas em uma pilha ou uma fila?

★ **Exercício R15.12.** Suponha que as strings de `"A" . . . "Z"` sejam inseridas em uma pilha. Depois, elas são removidas da pilha e colocadas sobre uma segunda pilha. Por fim, todas são tiradas da segunda pilha e impressas. Em que ordem as strings são impressas?

Exercícios de revisão adicionais estão disponíveis no WileyPLUS (recurso da editora original).

EXERCÍCIOS DE PROGRAMAÇÃO

★★ **Exercício P15.1.** Utilizando somente a interface pública da classe lista encadeada, escreva um método

```
public static void downsize(LinkedList<String> staff)
```

que remove os empregados alternadamente (um sim, um não) de uma lista encadeada.

★★ **Exercício P15.2.** Utilizando somente a interface pública da classe lista encadeada, escreva um método

```
public static void reverse(LinkedList<String> staff)
```

que inverte as entradas em uma lista encadeada.

★★★ **Exercício P15.3.** Adicione um método `reverse` à nossa implementação da classe `LinkedList` que inverte os links em uma lista. Implemente esse método redirecionando diretamente os links, sem utilizar um iterador.

★ **Exercício P15.4.** Adicione um método `size` a nossa implementação da classe `LinkedList` que calcula o número de elementos na lista, seguindo links e contando os elementos até o fim da lista ser alcançado.

★ **Exercício P15.5.** Adicione um campo `currentSize` à nossa implementação da classe `LinkedList`. Modifique os métodos `add` e `remove` da lista encadeada e do iterador de lista para atualizar o campo `currentSize` de modo que ele sempre contenha o tamanho correto. Altere o método `size` do exercício anterior de modo que retorne apenas o valor dessa variável de instância.

★★ **Exercício P15.6.** A classe lista encadeada da biblioteca padrão tem um método `add` que permite inserção eficiente no fim da lista. Implemente esse método na classe `LinkedList` na Seção 15.2. Adicione um campo de instância à classe lista encadeada que aponta para o último nó na lista. Certifique-se de que os outros métodos atualizem esse campo.

★★★ **Exercício P15.7.** Repita o Exercício P15.6, mas utilize uma estratégia de implementação diferente. Remova a referência ao primeiro nó na classe `LinkedList` e faça a referência

`next` do último nó apontar para o primeiro, de modo que todos os nós formem um ciclo. Tal implementação é chamada de *lista encadeada circular*.

★★★ **Exercício P15.8.** Reimplemente a classe `LinkedList` na Seção 15.2 de modo que as classes `Node` e `LinkedListIterator` não sejam classes internas.

★★★ **Exercício P15.9.** Adicione um campo `previous` à classe `Node` na Seção 15.2 e forneça os métodos `previous` e `hasPrevious` no iterador.

★★ **Exercício P15.10.** A biblioteca Java padrão implementa uma classe `Stack`, mas neste exercício solicitamos que você forneça sua própria implementação. Não implemente parâmetros de tipo. Utilize um array `Object[]` para armazenar os elementos da pilha. Quando o array se encher, aloque um array duas vezes maior e copie os valores para esse array.

★ **Exercício P15.11.** Implemente uma classe `Stack` utilizando uma lista encadeada para armazenar os elementos. Não implemente parâmetros de tipo.

★★ **Exercício P15.12.** Implemente uma fila como um *array circular* conforme mostrado a seguir: utilize duas variáveis de índice `head` (início da fila) e `tail` (final da fila) que contém o índice do próximo elemento a ser removido e o próximo elemento a ser adicionado. Depois que um elemento é removido ou adicionado, o índice é incrementado (veja Figura 14).

Após algum tempo, o elemento `tail` alcançará a parte superior do array. Então ele fará uma "volta" e recomeçará do 0 – veja Figura 15. Por essa razão, o array é chamado "circular".

```
public class CircularArrayQueue
{
   public CircularArrayQueue(int capacity) { . . . }
   public void add(Object x) { . . . }
   public Object remove() { . . . }
   public int size() { . . . }
   private int head;
   private int tail;
   private int theSize;
   private Object[] elements;
}
```

Essa implementação fornece uma fila *restrita* – cedo ou tarde ela enche. Veja o próximo exercício sobre como remover essa limitação.

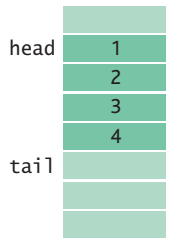


Figura 14
Adicionando e removendo elementos da fila.

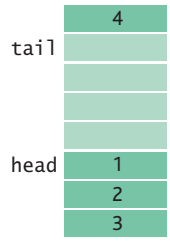


Figura 15
Uma fila que faz uma volta ao chegar ao fim do array.

★★★ **Exercício P15.13.** A fila do Exercício P15.12 pode ficar cheia se forem adicionados mais elementos do que o array pode armazenar. Aprimore a implementação conforme mostrado a seguir. Quando o array se encher, aloque um array maior, copie os valores para ele e atribua-o à variável de instância `elements`. *Dica:* você não pode simplesmente copiar os elementos para a mesma posição do novo array. Em vez disso, mova o elemento inicial para a posição 0.

★★ **Exercício P15.14.** Modifique o algoritmo de classificação por inserção do Tópico Avançado 14.1 para classificar uma lista encadeada.

★★ **Exercício P15.15.** Modifique a classe `Invoice` do Capítulo 12 de modo que ela implemente a interface `Iterable<LineItem>`. Então demonstre como um objeto `Invoice` pode ser utilizado em um laço "for each".

★★G **Exercício P15.16.** Escreva um programa para exibir uma lista encadeada graficamente. Desenhe cada elemento da lista como uma caixa e indique os links com segmentos de linha. Desenhe um iterador de acordo com a Figura 3. Forneça botões para mover o iterador e adicionar e remover elementos.

Exercícios de programação adicionais estão disponíveis no WileyPLUS (recurso da editora original).

PROJETOS DE PROGRAMAÇÃO

★★★ **Projeto 15.1.** Implemente uma classe `Polynomial` que descreve um polinômio como

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

Armazene um polinômio como uma lista encadeada de termos. Um termo contém o coeficiente e a potência de x. Por exemplo, você armazenaria $p(x)$ como

$$(5, 10), (9, 7), (-1, 1), (-10, 0)$$

Forneça métodos para adicionar, multiplicar e imprimir polinômios e calcular a derivada de um polinômio.

★★★ **Projeto 15.2.** Torne a implementação de lista deste capítulo tão poderosa quanto a implementação da biblioteca Java. (Mas não implemente parâmetros de tipo.)

- Forneça iteração bidirecional.
- Torne `Node` uma classe interna estática.
- Implemente as interfaces padrão `List` e `ListIterator` e forneça os métodos ausentes. (*Dica:* você pode achar mais fácil estender `AbstractList` em vez de implementar todos os métodos `List` a partir do zero.)

★★★ **Projeto 15.3.** Implemente o seguinte algoritmo para a avaliação de expressões aritméticas.

Cada operador tem uma *precedência*. Os operadores `+` e `-` têm a precedência mais baixa, `*` e `/` têm precedência maior (e igual), e `^` (que, neste exercício, significa "elevar a uma potência") tem a precedência mais alta. Por exemplo,

```
3 * 4 ^ 2 + 5
```

deve significar o mesmo que

```
(3 * (4 ^ 2)) + 5
```

com um valor de 53.

Em seu algoritmo, utilize duas pilhas. Uma pilha armazena números, a outra, operadores. Quando você encontrar um número, insira-o na pilha de números. Quando encontrar um operador, insira-o na pilha de operadores se ele tiver precedência mais alta que a do operador no topo da pilha. Caso contrário, remova um operador da pilha de operadores, remova dois números da pilha de números e insira o resultado do cálculo na pilha de números. Repita até que o topo da pilha de operadores tenha precedência mais baixa. Ao fim da expressão, limpe a pilha da mesma maneira. Por exemplo, eis como a expressão 3 * 4 ^ 2 + 5 é avaliada:

	Expressão:	3 * 4 ^ 2 + 5		
1	Expressão restante:	* 4 ^ 2 + 5	Pilha de números 3	Pilha de operadores
2	Expressão restante:	4 ^ 2 + 5	Pilha de números 3	Pilha de operadores *
3	Expressão restante:	^ 2 + 5	Pilha de números 4 3	Pilha de operadores *
4	Expressão restante:	2 + 5	Pilha de números 4 3	Pilha de operadores ^ *
5	Expressão restante:	+ 5	Pilha de números 2 4 3	Pilha de operadores ^ *
6	Expressão restante:	+ 5	Pilha de números 16 3	Pilha de operadores *
7	Expressão restante:	5	Pilha de números 48	Pilha de operadores +
8	Expressão restante:		Pilha de números 5 48	Pilha de operadores +
9	Expressão restante:		Pilha de números 53	Pilha de operadores

Você deve aprimorar esse algoritmo para lidar com parênteses. Além disso, certifique-se de que subtrações e divisões sejam feitas na ordem correta. Por exemplo, `12 - 5 - 3` deve resultar em 4.

RESPOSTAS ÀS PERGUNTAS DE AUTOVERIFICAÇÃO DA APRENDIZAGEM

1. Sim, por duas razões. Você precisa armazenar as referências de nó, e cada nó é um objeto separado. (Há um gasto adicional, ou overhead, fixo para armazenar cada objeto na máquina virtual.)
2. Um índice do tipo inteiro pode ser utilizado para acessar qualquer localização do array.
3. Quando a lista está vazia, `first` é `null`. Um novo `Node` é alocado. Seu campo `data` é configurado como o objeto recém-inserido. Seu campo `next` é configurado como `null` porque `first` é `null`. O campo `first` é configurado como o novo nó. O resultado é uma lista encadeada de comprimento 1.
4. Ele aponta para o elemento à esquerda. Você pode ver isso rastreando a primeira chamada a `next`. Ele deixa `position` apontar para o primeiro nó.
5. Se `position` for `null`, devemos estar no início da lista e inserir um elemento requer atualizar a referência `first`. Se estivermos no meio da lista, a referência `first` não deve ser alterada.
6. Você pode se concentrar nas características essenciais do tipo de dados sem se distrair com os detalhes da implementação.
7. A visão abstrata seria semelhante à Figura 9, mas com setas em ambas as direções. A visão concreta seria como a Figura 8, mas com referências ao nó anterior adicionadas a cada nó.
8. Para localizar o elemento intermediário, são necessários n / 2 passos. Para localizar o meio do subintervalo à esquerda ou direita, são necessários n / 4 passos. A próxima pesquisa leva n / 8 passos. Portanto, esperamos quase n passos para localizar um elemento. Nesse ponto, é melhor simplesmente fazer uma pesquisa linear que, em média, leva n / 2 passos.
9.

10. As pilhas utilizam a disciplina "último a entrar, primeiro a sair". Se você for o primeiro a enviar um trabalho de impressão e muitas pessoas adicionarem pedidos de impressão antes de a impressora ter uma chance de lidar com a sua, as cópias deles serão impressas primeiro e você terá de esperar até que todas fiquem prontas.

Apêndice A

Guia de Estilo para Codificação em Java

Introdução

Este guia de estilo de codificação é uma versão simplificada de um outro que tem sido utilizado com bastante sucesso tanto na indústria como em cursos universitários.

Um guia de estilo é um conjunto de requisitos obrigatórios para leiaute e formatação. Um estilo uniforme torna mais fácil a leitura de código descrito pelo seu professor e colegas. Você realmente apreciará isso se trabalhar em equipe. Também se torna mais fácil para seu professor e seus colegas entenderem rapidamente a essência de seus programas.

Um guia de estilo o torna um programador mais produtivo porque *reduz escolhas gratuitas.* Se não tiver de fazer escolhas sobre questões triviais, você poderá investir sua energia na solução de problemas reais.

Nessas diretrizes, várias construções são simplesmente expurgadas. Isso não significa que programadores que as utilizem sejam ruins ou incompetentes. Significa que essas construções não são essenciais e podem ser expressas de maneira igualmente adequada ou ainda melhor com outras construções da linguagem.

Se você já possui experiência em programação, em Java ou em outra linguagem, talvez, inicialmente, você se sinta desconfortável em abrir mão de alguns hábitos. Mas é um sinal de profissionalismo separar preferências pessoais de questões menores e chegar a um acordo para o benefício do seu grupo.

Essas diretrizes são necessariamente monótonas. Elas também mencionam recursos que talvez você ainda não tenha visto em aula. Eis os destaques mais importantes:

- Tabulações são configuradas a cada três espaços.
- Nomes de variáveis e de métodos estão em letras minúsculas, com alguns caracteres em letras maiúsculas no meio.
- Nomes de classes iniciam com uma letra maiúscula.
- Nomes de constantes estão em LETRAS MAIÚSCULAS, com um SUBLINHADO ocasional.
- Há espaços depois das palavras-chave e em volta dos operadores binários.
- As chaves precisam estar alinhadas horizontal ou verticalmente.
- Nenhum número mágico pode ser utilizado.
- Todos os métodos, exceto *main* e métodos de biblioteca sobrescritos, devem ter um comentário.
- No máximo 30 linhas de código podem ser utilizadas por método.
- `continue` e `break` não são permitidos.
- Todas as variáveis não-`final` devem ser privadas.

Observação para o professor: naturalmente, muitos programadores e organizações têm forte apego a um estilo de codificação. Se esse guia de estilo for incompatível com suas preferências ou costumes locais, sinta-se livre para modificá-lo. Para esse propósito, o guia de estilo de codificação está disponível na forma eletrônica no WileyPLUS ou no site da editora original `http://www.wiley.com/college/horstmann`.

Arquivos-fonte

Todo programa Java é uma coleção de um ou mais arquivos-fonte. O programa executável é obtido compilando-se esses arquivos. Organize o material em cada arquivo assim:

- instrução `package`, se apropriado
- instruções `import`
- Um comentário que explica o propósito desse arquivo
- Uma classe `public`
- Outras classes, se apropriado

O comentário que explica o propósito desse arquivo deve estar no formato reconhecido pelo utilitário `javadoc`. Inicie com um `/**` e utilize as marcas `@author` e `@version`:

```
/**
  COPYRIGHT (C) 2005 Harry Hacker. Todos os direitos reservados.
  Classes para manipular widgets.
  Resolve o exercício #3 do trabalho de casa CS101
  @author Harry Hacker
  @version 1.01 2005-02-15
*/
```

Classes

Cada classe deve ser precedida por um comentário de classe explicando seu propósito.

Primeiro liste todos os recursos públicos e depois todos os recursos privados.

Dentro das seções públicas e privadas, utilize a seguinte ordem:

1. Construtores
2. Métodos de instância
3. Métodos estáticos
4. Campos de instância
5. Campos estáticos
6. Classes internas

Deixe uma linha em branco após cada método.

Todas as variáveis não-`final` devem ser privadas. (Contudo, variáveis de instância de uma classe interna privada podem ser públicas.) Métodos e variáveis `final` podem ser públicos ou privados, conforme for apropriado.

Todos os recursos devem ser marcados como `public` ou `private`. Não utilize a visibilidade padrão (isto é, a visibilidade de pacote) nem o atributo `protected`.

Evite variáveis estáticas (exceto as `final`) sempre que possível. Na rara situação em que variáveis estáticas são necessárias, você tem permissão para usar uma variável estática por classe.

Métodos

Cada método (exceto `main`) inicia com um comentário no formato `javadoc`.

```
/**
   Converte a data de calendário em dia do calendário juliano.
   Note: esse algoritmo é da Press et al., Numerical Recipes
   in C, 2nd ed., Cambridge University Press, 1992.
   @param day dia a ser convertido
   @param month mês a ser convertido
   @param year ano a ser convertido
   @return número do dia juliano que inicia ao meio dia da
   data do calendário especificado.
*/
public static int getJulianDayNumber(int day, int month, int year)
{
   . . .
}
```

Os nomes dos parâmetros devem ser explícitos, especialmente se eles forem inteiros ou booleanos:

```
public Employee remove(int d, double s)
   // Hein?
public Employee remove(int department, double severancePay)
   // OK
```

Métodos devem ter no máximo 30 linhas de código. Assinatura, comentários, linhas em branco e linhas do método contendo apenas chaves não estão incluídos nessa contagem. Essa regra força a divisão dos cálculos complexos em métodos separados.

Variáveis e constantes

Não defina todas as variáveis no começo de um bloco:

```
{
   double xold; // Não faça isso
   double xnew;
   boolean more;
   . . .
}
```

Defina cada variável um pouco antes de ela ser utilizada pela primeira vez:

```
{
   . . .
   double xold = Integer.parseInt(input);
   boolean more = false;
   while (more)
   {
      double xnew = (xold + a / xold) / 2;
      . . .
   }
   . . .
}
```

Não defina duas variáveis na mesma linha:

```
int dimes = 0, nickels = 0; // Não faça isso
```

Em vez disso, utilize duas definições separadas:

```
int dimes = 0; // OK
int nickels = 0;
```

Em Java, constantes devem ser definidas com a palavra-chave `final`. Se a constante for utilizada por múltiplos métodos, declare-a como `static final`. Uma boa idéia é definir variáveis `final` estáticas como `private` se nenhuma outra classe tiver interesse nelas.

Não utilize *números mágicos!* Um número mágico é uma constante numérica predefinida no código, sem uma definição de constante. Qualquer número, exceto –1, 0, 1 e 2, é considerado mágico:

```
if (p.getX() < 300) // Não faça isso
```

Utilize em vez disso variáveis `final`:

```
final double WINDOW_WIDTH = 300;
. . .
if (p.getX() < WINDOW_WIDTH) // OK
```

Mesmo a constante cósmica mais razoável um belo dia irá mudar. Você acha que há 365 dias em um ano? Seus clientes em Marte ficarão muito magoados por causa desse ridículo preconceito. Crie uma constante

```
public static final int DAYS_PER_YEAR = 365;
```

assim, você pode produzir facilmente uma versão marciana sem tentar localizar todos os 365, 364, 366, 367 e assim por diante, no seu código.

Ao declarar variáveis de array, agrupe o `[ ]` com o tipo, não com a variável.

```
int[] values; // OK
int values[]; // Argh! – isso é um remanescente horroroso do C
```

Ao utilizar coleções, use parâmetros de tipo e não tipos "brutos".

```
ArrayList<String> names = new ArrayList<String>(); // OK
ArrayList names = new ArrayList(); // Inválido
```

Fluxo de controle

O comando `if`

Evite a armadilha "`if . . . if . . . else`". O código

```
if ( . . . )
   if ( . . . ) . . .;
else . . .;
```

não fará o que o nível de recuo sugere e pode demorar horas para encontrar esse bug. Sempre utilize um par extra de `{ . . . }` ao lidar com "`if . . . if . . . else`":

```
if ( . . . )
{
   if ( . . . ) . . .;
} // { . . . } são necessários
else . . .;
```

```
if ( . . . )
{
   if ( . . . ) . . .;
   else . . .;
} // { . . . } não necessários, mas eles o mantém longe de problemas
```

O comando `for`

Só utilize laços `for` quando uma variável variar de um valor a outro com um incremento/decremento constante:

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Ou, melhor ainda, utilize o laço "*for each*":

```
for (int e : a)
   System.out.println(e);
```

Não utilize o laço `for` para construções esquisitas como:

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew))
   // Não faça isso
```

Transforme esse laço em um laço `while`. Dessa maneira, a seqüência de instruções fica muito mais clara.

```
a = a / 2;
while (count < ITERATIONS) // OK
{
   . . .
   System.out.println(xnew);
}
```

Fluxo de controle não-linear

Evite o comando `switch`, porque na sucessão de casos é fácil cair acidentalmente em um caso indesejável. Utilize em vez disso `if/else`.

Evite as instruções `break` ou `continue`. Utilize outra variável `boolean` para controlar o fluxo de execução.

Exceções

Não marque um método com uma especificação de exceção excessivamente geral:

```
Widget readWidget(Reader in) throws Exception // Ruim
```

Em vez disso, declare especificamente todas as exceções verificadas que seu método pode lançar:

```
Widget readWidget(Reader in)
      throws IOException, MalformedWidgetException // Bom
```

Não "silencie" as exceções:

```
try
{
   double price = in.readDouble();
}
catch (Exception e)
{ } // Ruim
```

Iniciantes freqüentemente cometem esse erro de "manter o compilador feliz". Se o método atual não for apropriado para tratar a exceção, simplesmente utilize uma especificação `throws` e deixe que um dos chamadores trate essa exceção.

Questões léxicas

Convenções para atribuição de nomes

As regras a seguir especificam quando utilizar letras maiúsculas e minúsculas nos nomes de identificadores.

- Todos os nomes de variável e método e todos os campos de dados de classes são em letras minúsculas (talvez com uma letra maiúscula ocasional no meio); por exemplo, `firstPlayer`.
- Todas as constantes são em letras maiúsculas (talvez com um SUBLINHADO ocasional); por exemplo, `CLOCK_RADIUS`.

- Todos os nomes de classe e interface iniciam com letras maiúsculas e são seguidos por letras minúsculas (talvez com uma letra maiúscula ocasional); por exemplo, `BankTeller`.
- Variáveis de tipos genéricos estão em letras maiúsculas, normalmente uma única letra.

Os nomes devem ser razoavelmente longos e descritivos. Utilize `firstPlayer` em vez de `fp`. Nada de dscrtr vgs. Variáveis locais comumente utilizadas podem ser curtas (`ch`, `i`) desde que sejam apenas simples contêineres temporários para um caractere de entrada, contador de laço, etc. Além disso, não utilize `ctr`, `c`, `cntr`, `cnt` e `c2` para variáveis no seu método. Certamente todas essas variáveis têm propósitos específicos e podem ser nomeadas para que o leitor se lembre delas (por exemplo, `current`, `next`, `previous`, `result`, `. . .`) Mas é costume utilizar nomes de única letra, como `T` ou `E` para tipos genéricos.

Recuo e espaços em branco

Utilize paradas de tabulação a cada três colunas. Isso significa que você precisará mudar a configuração das paradas de tabulação no seu editor!

Utilize linhas em branco livremente para separar partes de um método que são logicamente distintas.

Utilize um espaço em branco em torno de cada operador binário:

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
// Bom
x1=(-b-Math.sqrt(b*b-4*a*c))/(2*a);
// Ruim
```

Deixe um espaço em branco após (e não antes de) cada vírgula ou ponto-e-vírgula. Não deixe um espaço antes ou depois de um parêntese ou colchete em uma expressão. Deixe espaços em torno da parte `( . . . )` de um comando `if`, `while`, `for` ou `catch`.

```
if (x == 0) y = 0;
f(a, b[i]);
```

Cada linha deve caber em 80 colunas. Se precisar dividir uma instrução, adicione um nível de recuo para a continuação:

```
a[n] = ...................................................
       + .................;
```

Inicie a linha recuada com um operador (se possível).

Se a condição em um comando `if` ou `while` precisar ser dividida, certifique-se de adicionar chaves em torno do corpo, mesmo se ele só consistir em uma instrução:

```
if ( ...................................................
     && .................
     || .......... )
{
   . . .
}
```

Se você não utilizar chaves, será difícil separar visualmente a continuação da condição da instrução a ser executada.

Chaves

As chaves de abertura e fechamento precisam estar alinhadas, horizontal ou verticalmente:

```
while (i < n) { System.out.println(a[i]); i++; }
```

```
while (i < n)
{
   System.out.println(a[i]);
   i++;
}
```

Alguns programadores não alinham as chaves verticais, mas posicionam a { atrás da palavra-chave:

```
while (i < n) { // NÃO FAÇA ISSO
   System.out.println(a[i]);
   i++;
}
```

Isso dificulta verificar se as chaves se correspondem.

Leiaute instável

Alguns programadores orgulham-se de alinhar certas colunas em seu código:

```
firstRecord  = other.firstRecord;
lastRecord   = other.lastRecord;
cutoff       = other.cutoff;
```

Inegavelmente isso é elegante, mas o leiaute não é estável sob alterações. Um novo nome de variável mais longo que o número de colunas pré-alocadas exige que você mova todas as entradas:

```
firstRecord  = other.firstRecord;
lastRecord   = other.lastRecord;
cutoff       = other.cutoff;
marginalFudgeFactor = other.marginalFudgeFactor;
```

Isso é apenas o tipo de armadilha que faz você decidir por utilizar um nome de variável curto como `mff`. Utilize um leiaute simples que seja fácil de manter à medida que seus programas mudam.

Apêndice B

Os Subconjuntos Basic Latin e Latin-1 do Unicode

Este apêndice lista os caracteres Unicode que são mais comumente utilizados para processar idiomas da Europa ocidental. Uma lista completa dos caracteres Unicode pode ser encontrada em `http://unicode.org`.

Tabela 1 Alguns caracteres de controle

Caractere	Código	Decimal	Seqüência de escape
Tabulação	`'\u0009'`	9	`'\t'`
Nova linha	`'\u000A'`	10	`'\n'`
Return	`'\u000D'`	13	`'\r'`
Espaço	`'\u0020'`	32	

Tabela 2 Subconjunto Basic Latin (ASCII) do Unicode

Caract.	Código	Dec.	Caract.	Código	Dec.	Caract.	Código	Dec.
			@	'\u0040'	64	`	'\u0060'	96
!	'\u0021'	33	A	'\u0041'	65	a	'\u0061'	97
"	'\u0022'	34	B	'\u0042'	66	b	'\u0062'	98
#	'\u0023'	35	C	'\u0043'	67	c	'\u0063'	99
$	'\u0024'	36	D	'\u0044'	68	d	'\u0064'	100
%	'\u0025'	37	E	'\u0045'	69	e	'\u0065'	101
&	'\u0026'	38	F	'\u0046'	70	f	'\u0066'	102
'	'\u0027'	39	G	'\u0047'	71	g	'\u0067'	103
(	'\u0028'	40	H	'\u0048'	72	h	'\u0068'	104
)	'\u0029'	41	I	'\u0049'	73	i	'\u0069'	105
*	'\u002A'	42	J	'\u004A'	74	j	'\u006A'	106
+	'\u002B'	43	K	'\u004B'	75	k	'\u006B'	107
,	'\u002C'	44	L	'\u004C'	76	l	'\u006C'	108
-	'\u002D'	45	M	'\u004D'	77	m	'\u006D'	109
.	'\u002E'	46	N	'\u004E'	78	n	'\u006E'	110
/	'\u002F'	47	O	'\u004F'	79	o	'\u006F'	111
0	'\u0030'	48	P	'\u0050'	80	p	'\u0070'	112
1	'\u0031'	49	Q	'\u0051'	81	q	'\u0071'	113
2	'\u0032'	50	R	'\u0052'	82	r	'\u0072'	114
3	'\u0033'	51	S	'\u0053'	83	s	'\u0073'	115
4	'\u0034'	52	T	'\u0054'	84	t	'\u0074'	116
5	'\u0035'	53	U	'\u0055'	85	u	'\u0075'	117
6	'\u0036'	54	V	'\u0056'	86	v	'\u0076'	118
7	'\u0037'	55	W	'\u0057'	87	w	'\u0077'	119
8	'\u0038'	56	X	'\u0058'	88	x	'\u0078'	120
9	'\u0039'	57	Y	'\u0059'	89	y	'\u0079'	121
:	'\u003A'	58	Z	'\u005A'	90	z	'\u007A'	122
;	'\u003B'	59	[	'\u005B'	91	{	'\u007B'	123
<	'\u003C'	60	\	'\u005C'	92	|	'\u007C'	124
=	'\u003D'	61	]	'\u005D'	93	}	'\u007D'	125
>	'\u003E'	62	^	'\u005E'	94	~	'\u007E'	126
?	'\u003F'	63	_	'\u005F'	95			

Tabela 3 Subconjunto Latin-1 do Unicode

Caract.	Código	Dec.	Caract.	Código	Dec.	Caract.	Código	Dec.
			À	'\u00C0'	192	à	'\u00E0'	224
¡	'\u00A1'	161	Á	'\u00C1'	193	á	'\u00E1'	225
¢	'\u00A2'	162	Â	'\u00C2'	194	â	'\u00E2'	226
£	'\u00A3'	163	Ã	'\u00C3'	195	ã	'\u00E3'	227
¤	'\u00A4'	164	Ä	'\u00C4'	196	ä	'\u00E4'	228
¥	'\u00A5'	165	Å	'\u00C5'	197	å	'\u00E5'	229
¦	'\u00A6'	166	Æ	'\u00C6'	198	æ	'\u00E6'	230
§	'\u00A7'	167	Ç	'\u00C7'	199	ç	'\u00E7'	231
¨	'\u00A8'	168	È	'\u00C8'	200	è	'\u00E8'	232
©	'\u00A9'	169	É	'\u00C9'	201	é	'\u00E9'	233
ª	'\u00AA'	170	Ê	'\u00CA'	202	ê	'\u00EA'	234
«	'\u00AB'	171	Ë	'\u00CB'	203	ë	'\u00EB'	235
¬	'\u00AC'	172	Ì	'\u00CC'	204	ì	'\u00EC'	236
-	'\u00AD'	173	Í	'\u00CD'	205	í	'\u00ED'	237
®	'\u00AE'	174	Î	'\u00CE'	206	î	'\u00EE'	238
¯	'\u00AF'	175	Ï	'\u00CF'	207	ï	'\u00EF'	239
°	'\u00B0'	176	Ð	'\u00D0'	208	ð	'\u00F0'	240
±	'\u00B1'	177	Ñ	'\u00D1'	209	ñ	'\u00F1'	241
²	'\u00B2'	178	Ò	'\u00D2'	210	ò	'\u00F2'	242
³	'\u00B3'	179	Ó	'\u00D3'	211	ó	'\u00F3'	243
´	'\u00B4'	180	Ô	'\u00D4'	212	ô	'\u00F4'	244
µ	'\u00B5'	181	Õ	'\u00D5'	213	õ	'\u00F5'	245
¶	'\u00B6'	182	Ö	'\u00D6'	214	ö	'\u00F6'	246
·	'\u00B7'	183	×	'\u00D7'	215	÷	'\u00F7'	247
¸	'\u00B8'	184	Ø	'\u00D8'	216	ø	'\u00F8'	248
¹	'\u00B9'	185	Ù	'\u00D9'	217	ù	'\u00F9'	249
º	'\u00BA'	186	Ú	'\u00DA'	218	ú	'\u00FA'	250
»	'\u00BB'	187	Û	'\u00DB'	219	û	'\u00FB'	251
¼	'\u00BC'	188	Ü	'\u00DC'	220	ü	'\u00FC'	252
½	'\u00BD'	189	Ý	'\u00DD'	221	ý	'\u00FD'	253
¾	'\u00BE'	190	Þ	'\u00DE'	222	þ	'\u00FE'	254
¿	'\u00BF'	191	ß	'\u00DF'	223	ÿ	'\u00FF'	255

C Apêndice

A Biblioteca Java

Este apêndice lista todas as classes e métodos da biblioteca Java padrão que são utilizados neste livro (inclusive nos capítulos disponíveis na Web).

Na hierarquia de herança a seguir, as superclasses que não são utilizadas neste livro são mostradas entre parênteses. Algumas classes implementam interfaces não-abordadas neste livro; elas são omitidas. As classes são classificadas primeiro por pacote, depois alfabeticamente dentro do pacote. As interfaces estão em itálico.

```
java.awt.Shape
java.lang.Cloneable
java.lang.Object
   java.awt.BorderLayout
   java.awt.Color
   java.awt.Component
      java.awt.Container
         javax.swing.JComponent
            javax.swing.AbstractButton
               javax.swing.JButton
               javax.swing.JMenuItem
                  javax.swing.JMenu
              (javax.swing.JToggleButton)
                  javax.swing.JCheckBox
                  javax.swing.JRadioButton
            javax.swing.JComboBox
            javax.swing.JFileChooser
            javax.swing.JLabel
            javax.swing.JMenuBar
            javax.swing.JPanel
            javax.swing.JOptionPane
            javax.swing.JScrollPane
            javax.swing.JSlider
```

```
            javax.swing.text.JTextComponent
               javax.swing.JTextArea
               javax.swing.JTextField
            (java.awt.Panel)
            java.applet.Applet
               javax.swing.JApplet
   (java.awt.Window)
      java.awt.Frame
         javax.swing.JFrame
(java.awt.Dimension2D)
   java.awt.Dimension
 java.awt.FlowLayout
 java.awt.Font
 java.awt.Graphics
   java.awt.Graphics2D;
 java.awt.GridLayout
 java.awt.event.MouseAdapter implements MouseListener
 java.awt.geom.Line2D implements Cloneable, Shape
   java.awt.geom.Line2D.Double
 java.awt.geom.Point2D implements Cloneable
   java.awt.geom.Point2D.Double
 java.awt.geom.RectangularShape implements Cloneable, Shape
   (java.awt.geom.Rectangle2D)
      java.awt.Rectangle
   java.awt.geom.Ellipse2D
      java.awt.geom.Ellipse2D.Double
 java.io.File implements Comparable
 java.io.InputStream
   java.io.FileInputStream
   java.io.ObjectInputStream
 java.io.OutputStream
   java.io.FileOutputStream
   (java.io.FilterOutputStream)
      java.io.PrintStream
   java.io.ObjectOutputStream
 java.io.RandomAccessFile
 java.io.Reader
   (java.io.InputStreamReader)
      java.io.FileReader
 java.io.Writer
   java.io.PrintWriter
   (java.io.OutputStreamWriter)
      java.io.FileWriter
 java.lang.Boolean
 java.lang.Character implements Comparable
 java.lang.Math
(java.lang.Number)
   java.math.BigDecimal implements Comparable
   java.math.BigInteger implements Comparable
   java.lang.Double implements Comparable
   java.lang.Integer implements Comparable
 java.lang.String implements Comparable
 java.lang.System
 java.lang.Throwable
   java.lang.Error
   java.lang.Exception
      java.lang.CloneNotSupportedException
      java.io.IOException
         java.io.EOFException
         java.io.FileNotFoundException
```

```
        java.lang.RuntimeException
           java.lang.IllegalArgumentException
              java.lang.NumberFormatException
           java.lang.IllegalStateException
           java.util.NoSuchElementException
              java.util.InputMismatchException
           java.lang.NullPointerException
  (java.util.AbstractCollection<E>)
      java.util.AbstractList<E>
        (java.util.AbstractSequentialList<E>)
            java.util.LinkedList<E>
         java.util.ArrayList<E> implements Cloneable, List<E>
     (java.util.AbstractQueue<E>)
         java.util.PriorityQueue<E>
     (java.util.AbstractSet<E>)
         java.util.HashSet<E> implements Cloneable, Set<E>
         java.util.TreeSet<E> implements Cloneable, SortedSet<E>
  (java.util.AbstractMap<K, V>)
      java.util.HashMap<K, V> implements Cloneable, Map<K, V>
      java.util.TreeMap<K, V> implements Cloneable, SortedMap<K, V>
   java.util.Arrays
   java.util.Collections
   java.util.Calendar
      java.util.GregorianCalendar
   java.util.EventObject
     (java.awt.AWTEvent)
         java.awt.event.ActionEvent
        (java.awt.event.ComponentEvent)
           (java.awt.event.InputEvent)
               java.awt.event.MouseEvent
      javax.swing.event.ChangeEvent
   java.util.Random
   java.util.Scanner
   java.util.logging.Level
   java.util.logging.Logger
   javax.swing.ButtonGroup
   javax.swing.ImageIcon
   javax.swing.Timer
  (javax.swing.border.AbstractBorder )
      javax.swing.border.EtchedBorder
      javax.swing.border.TitledBorder
java.lang.Comparable<T>
java.util.Collection<E>
   java.util.List<E>
   java.util.Set<E>
      java.util.SortedSet<E>
java.util.Comparator<T>
(java.util.EventListener)
   java.awt.event.ActionListener
   java.awt.event.MouseListener
   javax.swing.event.ChangeListener
java.util.Iterator<E>
   java.util.ListIterator<E>
java.util.Map<K, V>
   java.util.SortedMap<K, V>
```

Nas descrições a seguir, a frase “esse objeto” (“esse componente”, “esse contêiner”, etc.) significa o objeto (componente, contêiner, etc.) sobre o qual o método é invocado (o parâmetro implícito `this`).

Pacote `java.applet`

Classe `java.applet.Applet`

- `void destroy()`
  Esse método é chamado quando a applet está para ser terminada, depois da última chamada a `stop`.
- `void init()`
  Esse método é chamado quando a applet foi carregada, antes da primeira chamada a `start`. As applets sobrescrevem esse método para executar a inicialização específica à applet e ler parâmetros de applet.
- `void start()`
  Esse método é chamado depois do método `init` e toda vez que a applet é revisitada.
- `void stop()`
  Esse método é chamado sempre que o usuário parar de usar essa applet.

Pacote `java.awt`

Classe `java.awt.BorderLayout`

- `BorderLayout()`
  Constrói um leiaute de borda. Um leiaute de borda tem cinco regiões para adicionar componentes, chamadas `"North"`, `"East"`, `"South"`, `"West"` e `"Center"`.
- `static final int CENTER`
  Esse valor identifica a posição de centro de um leiaute de borda.
- `static final int EAST`
  Esse valor identifica a posição leste de um leiaute de borda.
- `static final int NORTH`
  Esse valor identifica a posição norte de um leiaute de borda.
- `static final int SOUTH`
  Esse valor identifica a posição sul de um leiaute de borda.
- `static final int WEST`
  Esse valor identifica a posição oeste de um leiaute de borda.

Classe `java.awt.Color`

- `Color(float red, float green, float blue)`
  Cria uma cor com os valores vermelho, verde e azul especificados entre `0.0F` e `1.0F`.
  Parâmetros: `red` Componente vermelho
  `green` Componente verde
  `blue` Componente azul

Classe `java.awt.Component`

- `void addMouseListener(MouseListener listener)`
  Esse método adiciona um ouvinte de mouse ao componente.
  Parâmetros: `listener` Ouvinte de mouse a ser adicionado
- `int getHeight()`
  Esse método obtém a altura desse componente.
  Retorna: Altura em pixels.
- `int getWidth()`
  Esse método obtém a largura desse componente.
  Retorna: Largura em pixels.
- `void repaint()`
  Esse método repinta esse componente agendando uma chamada ao método `paint`.
- `void setPreferredSize(Dimension preferredSize)`
  Esse método configura o tamanho preferido desse componente.
- `void setSize(int width, int height)`
  Esse método configura o tamanho desse componente.
  Parâmetros: `width` Largura do componente
  `height` Altura do componente
- `void setVisible(boolean visible)`
  Esse método mostra ou oculta o componente.
  Parâmetros: `visible` `true` para mostrar o componente ou `false` para ocultá-lo

Classe `java.awt.Container`

- `void add(Component c)`
- `void add(Component c, Object position)`
  Esses métodos adicionam um componente ao fim desse contêiner. Se uma posição é dada, o gerenciador de leiaute é chamado paa posicionar o componente.
  Parâmetros: `c` Componente a ser adicionado
  `position` Um objeto que expressa as informações de posição para o gerenciador de leiaute
- `void setLayout(LayoutManager manager)`
  Esse método configura o gerenciador de leiaute para esse contêiner.
  Parâmetros: `manager` Um gerenciador de leiaute

Classe `java.awt.Dimension`

- `Dimension(int width, int height)`
  Constrói um objeto `Dimension` com a largura e a altura dadas.
  Parâmetros: `width` Largura
  `height` Altura

Classe `java.awt.FlowLayout`

- `FlowLayout()`

Constrói um novo leiaute de fluxo. Um leiaute de fluxo coloca o maior número de componentes possível em uma linha, sem mudar o tamanho deles e inicia novas linhas quando necessário.

Classe `java.awt.Font`

- `Font(String name, int style, int size)`

  Constrói um objeto fonte a partir do nome, estilo e tamanho em pontos especificados.

  Parâmetros: `name` Nome da fonte, que tanto pode ser o nome da família da fonte como o nome lógico da fonte, que deve ser um entre `"Dialog"`, `"DialogInput"`, `"Monospaced"`, `"Serif"` e `"SansSerif"`

  `style` Um entre `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` e `Font.ITALIC+Font.BOLD`

  `size` Tamanho em pontos da fonte

Classe `java.awt.Frame`

- `void setTitle(String title)`

  Esse método configura o título do frame.

  Parâmetros: `title` Título a ser exibido na borda do frame

Classe `java.awt.Graphics`

- `void setColor(Color c)`

  Esse método configura a cor atual. De agora em diante, todas as operações gráficas utilizam essa cor.

  Parâmetros: `c` Nova cor do desenho

Classe `java.awt.Graphics2D`

- `void draw(Shape s)`

  Esse método desenha o contorno da forma dada. Muitas classes – entre elas `Rectangle` e `Line2D.Double` – implementam a interface `Shape`.

  Parâmetros: `s` Forma a ser desenhada

- `void drawString(String s, int x, int y)`
- `void drawString(String s, float x, float y)`

  Esses métodos desenham uma string na fonte atual.

  Parâmetros: `s` String a desenhar

  `x,y` Ponto de base do primeiro caractere na string

- `void fill(Shape s)`

  Esse método desenha a forma dada e preenche com a cor atual.

  Parâmetros: `s` Forma a ser preenchida

Classe `java.awt.GridLayout`

- `GridLayout(int rows, int cols)`
  Esse construtor cria um leiaute de grade com o número especificado de linhas e colunas. Os componentes em um leiaute de grade são organizados em uma grade com largura e altura iguais. Uma, mas não ambas, entre `rows` e `cols` pode ser zero, caso em que qualquer número de objetos pode ser colocado em uma linha ou coluna, respectivamente.
  Parâmetros: `rows` Número de linhas na grade
  `cols` Número de colunas na grade

Classe `java.awt.Rectangle`

- `Rectangle()`
  Constrói um retângulo cujo canto superior esquerdo está em (0, 0) e cuja largura e altura são, ambas, zero.
- `Rectangle(int x, int y, int width, int height)`
  Constrói um retângulo com o canto superior esquerdo e o tamanho determinados.
  Parâmetros: `x,y` Canto superior-esquerdo
  `width` Largura
  `height` Altura
- `double getHeight()`
- `double getWidth()`
  Esses métodos obtêm a altura e a largura do retângulo.
- `double getX()`
- `double getY()`
  Esses métodos obtêm as coordenadas *x* e *y* do canto superior esquerdo do retângulo.
- `Rectangle intersection(Rectangle other)`
  Esse método calcula a intersecção desse retângulo com o retângulo especificado.
  Parâmetros: `other` Um retângulo
  Retorna: O maior retângulo contido em ambos `this` e `other`
- `void setLocation(int x, int y)`
  Esse método move esse retângulo para uma nova localização.
  Parâmetros: `x,y` Novo canto superior esquerdo
- `void translate(int dx, int dy)`
  Esse método move esse retângulo.
  Parâmetros: `dx` Distância a mover ao longo do eixo *x*
  `dy` Distância a mover ao longo do eixo *y*
- `Rectangle union(Rectangle other)`
  Esse método calcula a união desse retângulo com o retângulo especificado. Essa não é a união da teoria dos conjuntos, mas o menor retângulo que contém `this` e `other`.
  Parâmetros: `other` Um retângulo
  Retorna: O menor retângulo contendo `this` e `other`

Interface `java.awt.Shape`

A interface `Shape` descreve formas que podem ser desenhadas e preenchidas por um objeto `Graphics2D`.

Pacote `java.awt.event`

Interface `java.awt.event.ActionListener`

- `void actionPerformed(ActionEvent e)`
  A fonte do evento chama esse método quando ocorre uma ação.

Classe `java.awt.event.MouseEvent`

- `int getX()`
  Esse método retorna a posição horizontal do mouse no momento em que o evento ocorreu.
  Retorna: Coordenada posição *x* do mouse
- `int getY()`
  Esse método retorna a posição vertical do mouse no momento em que o evento ocorreu.
  Retorna: Coordenada *y* do mouse

Interface `java.awt.event.MouseListener`

- `void mouseClicked(MouseEvent e)`
  Esse método é chamado quando o mouse é clicado (isto é, pressionado e solto em sucessão rápida).
- `void mouseEntered(MouseEvent e)`
  Esse método é chamado quando o mouse está sobre o componente ao qual esse ouvinte foi adicionado.
- `void mouseExited(MouseEvent e)`
  Esse método é chamado quando o mouse saiu de cima do componente ao qual esse ouvinte foi adicionado.
- `void mousePressed(MouseEvent e)`
  Esse método é chamado quando um botão do mouse é pressionado.
- `void mouseReleased(MouseEvent e)`
  Esse método é chamado quando um botão do mouse é liberado.

Pacote `java.awt.geom`

Classe `java.awt.geom.Ellipse2D.Double`

- `Ellipse2D.Double(double x, double y, double w, double h)`
  Cria uma elipse a partir das coordenadas especificadas.
  Parâmetros: `x, y` Canto superior esquerdo do retângulo delimitador
  `w` Largura do retângulo delimitador
  `h` Altura do retângulo delimitador

Classe `java.awt.geom.Line2D`

- `double getX1()`
- `double getX2()`
- `double getY1()`
- `double getY2()`

  Esses métodos obtêm a coordenada solicitada de uma extremidade da linha.

  Retorna: Coordenada *x* ou *y* da primeira ou da segunda extremidade

- `void setLine(double x1, double y1, double x2, double y2)`

  Esse método configura as extremidades da linha.

  Parâmetros: `x1, y1` Uma nova extremidade dessa linha
  `x2, y2` A outra nova extremidade

Classe `java.awt.geom.Line2D.Double`

- `Line2D.Double(double x1, double y1, double x2, double y2)`

  Constrói uma linha a partir das coordenadas especificadas.

  Parâmetros: `x1, y1` Uma extremidade da linha
  `x2, y2` A outra extremidade

- `Line2D.Double(Point2D p1, Point2D p2)`

  Cria uma linha a partir das duas extremidades.

  Parâmetros: `p1, p2` Extremidades da linha

Classe `java.awt.geom.Point2D`

- `double getX()`
- `double getY()`

  Esses métodos obtêm as coordenadas desse ponto solicitadas.

  Retorna: Coordenadas *x* ou *y* desse ponto

- `void setLocation(double x, double y)`

  Esse método configura as coordenadas *x* e *y* desse ponto.

  Parâmetros: `x, y` Nova localização desse ponto

Classe `java.awt.geom.Point2D.Double`

- `Point2D.Double(double x, double y)`

  Constrói um ponto com as coordenadas especificadas.

  Parâmetros: `x, y` Coordenadas do ponto

Classe `java.awt.geom.RectangularShape`

- `int getHeight()`
- `int getWidth()`

  Esses métodos obtêm a altura ou largura do retângulo delimitador dessa forma retangular.

  Retorna: A altura ou a largura, respectivamente

- `double getCenterX()`
- `double getCenterY()`
- `double getMaxX()`
- `double getMaxY()`
- `double getMinX()`
- `double getMinY()`

  Esses métodos obtêm o valor das coordenadas solicitadas dos cantos ou do centro do retângulo delimitador dessa forma.

  Retorna: Coordenadas *x* e *y* do centro, do máximo ou do mínimo

Pacote `java.io`

Classe `java.io.EOFException`

- `EOFException(String message)`

  Constrói um objeto de exceção do tipo "fim de arquivo" (*end of file*).

  Parâmetros: `message` Mensagem detalhada

Classe `java.io.File`

- `File(String name)`

  Constrói um objeto `File` que descreve um arquivo (que pode ou não existir) com o nome dado.

  Parâmetros: `name` Nome do arquivo

- `boolean exists()`

  Esse método verifica se há um arquivo no sistema de arquivos local que corresponde a esse objeto `File`.

  Retorna: `true` se houver um arquivo correspondente, `false` caso contrário

Classe `java.io.FileInputStream`

- `FileInputStream(File f)`

  Constrói um fluxo de entrada de arquivo e abre o arquivo escolhido. Se o arquivo não pode ser aberto para leitura, uma exceção do tipo `FileNotFoundException` é lançada.

  Parâmetros: `f` Arquivo a ser aberto para leitura

- `FileInputStream(String name)`

  Constrói um fluxo de entrada de arquivo e abre o arquivo identificado. Se o arquivo não pode ser aberto para leitura, uma exceção do tipo `FileNotFoundException` é lançada.

  Parâmetros: `name` Nome do arquivo a ser aberto para leitura

Classe `java.io.FileNotFoundException`

Essa exceção é lançada quando um arquivo não pôde ser aberto.

Classe `java.io.FileOutputStream`

- `FileOutputStream(File f)`
  Constrói um fluxo de saída de arquivo e abre o arquivo escolhido. Se o arquivo não pode ser aberto para gravação, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `f` Arquivo a ser aberto para gravação
- `FileOutputStream(String name)`
  Constrói um fluxo de saída de arquivo e abre o arquivo identificado. Se o arquivo não pode ser aberto para gravação, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `name` Nome do arquivo a ser aberto para gravação

Classe `java.io.FileReader`

- `FileReader(File f)`
  Constrói um leitor de arquivo e abre o arquivo escolhido. Se o arquivo não pode ser aberto para leitura, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `f` Arquivo a ser aberto para leitura
- `FileReader(String name)`
  Constrói um leitor de arquivo e abre o arquivo identificado. Se o arquivo não pode ser aberto para leitura, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `name` Nome do arquivo a ser aberto para leitura

Classe `java.io.FileWriter`

- `FileWriter(File f)`
  Constrói um gravador de arquivo e abre o arquivo escolhido. Se o arquivo não pode ser aberto para gravação, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `f` Arquivo a ser aberto para gravação
- `FileWriter(String name)`
  Constrói um gravador de arquivo e abre o arquivo identificado. Se o arquivo não pode ser aberto para gravação, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `name` Nome do arquivo a ser aberto para gravação

Classe `java.io.InputStream`

- `void close()`
  Esse método fecha esse fluxo de entrada (como um `FileInputStream`) e libera todos os recursos do sistema associados com o fluxo.
- `int read()`
  Esse método lê o próximo byte de dados a partir desse fluxo de entrada.
  Retorna: O próximo byte de dados, ou -1 se o fim do fluxo for alcançado.

Classe `java.io.InputStreamReader`

- `InputStreamReader(InputStream in)`
  Constrói um leitor de um fluxo de entrada especificado.
  Parâmetros: `in` Fluxo de dados a ser lido

Classe `java.io.IOException`

Esse tipo de exceção é lançado quando um erro de entrada/saída é encontrado.

Classe `java.io.ObjectInputStream`

- `ObjectInputStream(InputStream in)`
  Constrói um objeto do tipo fluxo de entrada.
  Parâmetros: `in` Fluxo de dados a ser lido
- `Object readObject()`
  Esse método lê o próximo objeto a partir desse objeto fluxo de entrada.
  Retorna: Próximo objeto

Classe `java.io.ObjectOutputStream`

- `ObjectOutputStream(OutputStream out)`
  Constrói um objeto fluxo de saída.
  Parâmetros: `out` Fluxo a ser gravado em
- `Object writeObject(Object obj)`
  Esse método grava o próximo objeto para esse objeto fluxo de saída.
  Parâmetros: `obj` Objeto a ser gravado

Classe `java.io.OutputStream`

- `void close()`
  Esse método fecha esse fluxo de saída (como um `FileOutputStream`) e libera todos os recursos do sistema associados a esse fluxo. Um fluxo fechado não pode realizar operações de saída e não pode ser reaberto.
- `void write(int b)`
  Esse método grava o byte de ordem inferior de `b` nesse fluxo de saída.
  Parâmetros: `b` Inteiro cujo byte de ordem inferior é gravado

Classe `java.io.PrintStream`

- `void print(int x)`
- `void print(double x)`
- `void print(Object x)`
- `void print(String x)`
- `void println()`
- `void println(int x)`
- `void println(double x)`
- `void println(Object x)`
- `void println(String x)`

  Esses métodos imprimem um valor nesse fluxo de impressão. Os métodos `println` imprimem uma nova linha depois do valor. Os objetos são impressos convertendo-os em strings com seus métodos `toString`.
  Parâmetros: `x` Valor a ser impresso

- `PrintStream printf(Sting format, Object... values)`
  Esse método imprime a string formatada, substituindo os marcadores de lugar que começam com `%` com os valores dados.
  Parâmetros: `format` String de formato
  `values` Valores a serem impressos. Você pode fornecer qualquer quantidade de valores
  Retorna: Parâmetro implícito

Classe `java.io.PrintWriter`

- `PrintWriter(String name)`
  Constrói um gravador de impressão e abre o arquivo identificado. Se o arquivo não pode ser aberto para gravação, uma exceção do tipo `FileNotFoundException` é lançada.
  Parâmetros: `name` Nome do arquivo a ser aberto para gravação
- `PrintWriter(Writer out)`
  Constrói um gravador de impressão a partir de um gravador especificado (como um `FileWriter`).
  Parâmetros: `out` Gravador onde gravar a saída
- `void close()`
  Esse método fecha esse gravador e libera todos os recursos do sistema associados.
- `void print(int x)`
- `void print(double x)`
- `void print(Object x)`
- `void print(String x)`
- `void println()`
- `void println(int x)`
- `void println(double x)`
- `void println(Object x)`
- `void println(String x)`
  Esses métodos imprimem um valor para esse gravador de impressão. Os métodos `println` imprimem uma nova linha depois do valor. Os objetos são impressos convertendo-os em strings com seus métodos `toString`.
  Parâmetros: `x` Valor a ser impresso

Classe `java.io.RandomAccessFile`

- `RandomAccessFile(String name, String mode)`
  Esse método abre um arquivo de acesso aleatório identificado para acesso de leitura ou leitura/gravação.
  Parâmetros: `name` Nome do arquivo
  `mode` `"r"` para acesso de leitura ou `"rw"` para acesso de leitura/gravação

- `long getFilePointer()`
  Esse método obtém a posição atual nesse arquivo.
  Retorna: Posição atual para leitura e gravação
- `long length()`
  Esse método obtém o comprimento desse arquivo.
  Retorna: Comprimento do arquivo
- `char readChar()`
- `double readDouble()`
- `int readInt()`
  Esses métodos lêem um valor a partir da posição atual nesse arquivo.
  Retorna: Valor que foi lido do arquivo
- `void seek(long position)`
  Esse método configura a posição para leitura e gravação nesse arquivo.
  Parâmetros: `position` Nova posição
- `void writeChar(int x)`
- `void writeChars(String x)`
- `void writeDouble(double x)`
- `void writeInt(int x)`
  Esses métodos gravam um valor na posição atual desse arquivo.
  Parâmetros: `x` Valor a ser gravado

Classe `java.io.Reader`

- `void close()`
  Esse método fecha esse leitor e libera todos os recursos do sistema associados.
- `int read()`
  Esse método lê o próximo caractere desse leitor (como um `FileReader`).
  Retorna: Próximo caractere ou –1 se o fim da entrada for alcançado

Classe `java.io.Writer`

- `void close()`
  Esse método fecha esse gravador e libera todos os recursos do sistema associados.
- `void write(int b)`
  Esse método grava os dois bytes de ordem inferior de `b` nesse gravador (como um `FileWriter`).
  Parâmetros: `b` Inteiro cujos dois bytes de ordem inferior são gravados

Pacote `java.lang`

Classe `java.lang.Boolean`

- `Boolean(boolean value)`
  Constrói um objeto empacotador para um valor `boolean`.
  Parâmetros: `value` Valor a ser armazenado nesse objeto
- `boolean booleanValue()`
  Esse método retorna o valor `boolean` armazenado nesse objeto `Boolean`.
  Retorna: Valor booleano desse objeto

Classe `java.lang.Character`

- `static boolean isDigit(ch)`
  Esse método testa se um dado caractere é um dígito Unicode.
  Parâmetros: `ch` Caractere a ser testado
  Retorna: `true` se o caractere for um dígito
- `static boolean isLetter(ch)`
  Esse método testa se um dado caractere é uma letra Unicode.
  Parâmetros: `ch` Caractere a ser testado
  Retorna: `true` se o caractere for uma letra
- `static boolean isLowerCase(ch)`
  Esse método testa se um dado caractere é uma letra Unicode minúscula.
  Parâmetros: `ch` Caractere a ser testado
  Retorna: `true` se o caractere for uma letra minúscula
- `static boolean isUpperCase(ch)`
  Esse método testa se um dado caractere é uma letra Unicode maiúscula.
  Parâmetros: `ch` Caractere a ser testado
  Retorna: `true` se o caractere for uma letra maiúscula

Interface `java.lang.Cloneable`

Uma classe implementa essa interface para indicar que o método `Object.clone` tem permissão de fazer uma cópia superficial de suas variáveis de instância.

Classe `java.lang.CloneNotSupportedException`

Essa exceção é lançada quando um programa tenta utilizar `Object.clone` para fazer a cópia superficial de um objeto de uma classe que não implementa a interface `Cloneable`.

Interface `java.lang.Comparable<T>`

- `int compareTo(T other)`
  Esse método compara esse objeto com o objeto `other`.
  Parâmetros: `other` O objeto a ser comparado
  Retorna: Um inteiro negativo se esse objeto é menor que os outros, zero se são iguais ou um inteiro positivo caso contrário

Classe `java.lang.Double`

- `Double(double value)`
  Constrói um objeto empacotador para um número de ponto flutuante de dupla precisão.
  Parâmetros: `value` Valor a ser armazenado nesse objeto
- `double doubleValue()`
  Esse método retorna o valor de ponto flutuante armazenado nesse objeto empacotador `Double`.
  Retorna: Valor armazenado no objeto
- `static double parseDouble(String s)`
  Esse método retorna o número de ponto flutuante que a string representa. Se a string não pode ser interpretada como um número, uma `NumberFormatException` é lançada.
  Parâmetros: `s` String a ser analisada sintaticamente
  Retorna: Valor representado pelo parâmetro string

Classe `java.lang.Error`

Superclasse de todos os erros do sistema não-verificados.

Classe `java.lang.IllegalArgumentException`

- `IllegalArgumentException()`
  Constrói uma exceção do tipo `IllegalArgumentException` sem mensagem detalhada.

Classe `java.lang.IllegalStateException`

Essa exceção é lançada se o estado de um objeto indicar que um método não pode ser aplicado no momento.

Classe `java.lang.Integer`

- `Integer(int value)`
  Constrói um objeto empacotador para um inteiro.
  Parâmetros: `value` Valor a ser armazenado nesse objeto
- `int intValue()`
  Esse método retorna o valor inteiro armazenado nesse objeto empacotador.
  Retorna: Valor armazenado no objeto
- `static int parseInt(String s)`
  Esse método retorna o inteiro que a string representa. Se a string não pode ser interpretada como um inteiro, uma exceção do tipo `NumberFormatException` é lançada.
  Parâmetros: `s` String a ser analisada sintaticamente
  Retorna: Valor representado pelo parâmetro string
- `static Integer parseInt(String s, int base)`
  Esse método retorna o valor inteiro que a string representa em um dado sistema de números. Se a string não pode ser interpretada como um inteiro, uma exceção do tipo `NumberFormatException` é lançada.
  Parâmetros: `s` String a ser analisada sintaticamente
  `base` Base do sistema numérico (como 2 ou 16)
  Retorna: Valor representado pelo parâmetro string

- `static String toString(int i)`
- `static String toString(int i, int base)`

  Esse método cria uma representação de string de um inteiro em um dado sistema numérico. Se nenhuma base for fornecida, uma representação decimal é criada.

  Parâmetros: `i` Um número inteiro
  `base` Base do sistema numérico (como 2 ou 16)

  Retorna: Representação de string do parâmetro numérico no sistema numérico especificado

- `static final int MAX_VALUE`

  Essa constante é o maior valor do tipo `int`.

- `static final int MIN_VALUE`

  Essa constante é o menor valor (negativo) do tipo `int`.

Classe `java.lang.Math`

- `static double abs(double x)`

  Esse método retorna o valor absoluto $|x|$.

  Parâmetros: `x` Valor de ponto flutuante

  Retorna: Valor absoluto do parâmetro

- `static double acos(double x)`

  Esse método retorna o ângulo dado pelo arco cosseno, $\cos^{-1} x \in [0, \pi]$.

  Parâmetros: `x` Um valor de ponto flutuante entre –1 e 1

  Retorna: Arco cosseno do parâmetro, em radianos

- `static double asin(double x)`

  Esse método retorna o ângulo dado pelo arco seno, $\sin^{-1} x \in [-\pi/2, \pi/2]$.

  Parâmetros: `x` Um valor de ponto flutuante entre –1 e 1

  Retorna: Arco seno do parâmetro, em radianos

- `static double atan(double x)`

  Esse método retorna o ângulo dado pelo arco tangente, $\tan^{-1} x\ (-\pi/2, \pi/2)$.

  Parâmetros: `x` Um valor de ponto flutuante

  Retorna: Arco tangente do parâmetro, em radianos

- `static double atan2(double y, double x)`

  Esse método retorna o arco tangente, $\tan^{-1} (y/x) \in (-\pi, \pi)$. Se x for igual a zero ou se ele for necessário para distinguir "noroeste" de "sudeste" e "nordeste" de "sudoeste", utilize esse método em vez de `atan(y/x)`.

  Parâmetros: `y,x` Dois valores de ponto flutuante

  Retorna: Ângulo, em radianos, entre os pontos (0,0) e (x,y)

- `static double ceil(double x)`

  Esse método retorna o menor inteiro $\geq x$ (como um `double`).

  Parâmetros: `x` Um valor de ponto flutuante

  Retorna: Maior inteiro ("teto") do parâmetro

- `static double cos(double radians)`

  Esse método retorna o cosseno de um ângulo em radianos.

  Parâmetros: `radians` Um ângulo, em radianos

  Retorna: Cosseno do parâmetro

- `static double exp(double x)`
  Esse método retorna o valor e^x, onde e é a base dos logaritmos naturais.
  Parâmetros: `x` Um valor de ponto flutuante
  Retorna: e^x

- `static double floor(double x)`
  Esse método retorna o maior inteiro $\leq x$ (como um `double`).
  Parâmetros: `x` Um valor de ponto flutuante
  Retorna: Menor inteiro ("piso") do parâmetro

- `static double log(double x)`
  Esse método retorna o logaritmo natural (base e) de x, $\ln x$.
  Parâmetros: `x` Um número maior que 0.0
  Retorna: Logaritmo natural do parâmetro

- `static int max(int x, int y)`
- `static double max(double x, double y)`
  Esses métodos retornam o maior entre os dois valores dados como parâmetros.
  Parâmetros: `x, y` Dois valores inteiros ou de ponto flutuante
  Retorna: O máximo entre os valores de parâmetro

- `static int min(int x, int y)`
- `static double min(double x, double y)`
  Esses métodos retornam o menor entre os dois valores dados como parâmetro.
  Parâmetros: `x, y` Dois valores inteiros ou de ponto flutuante
  Retorna: O mínimo entre os valores de parâmetro

- `static double pow(double x, double y)`
  Esse método retorna o valor x^y ($x > 0$ ou $x = 0$ e $y > 0$ ou $x < 0$ e y é um inteiro).
  Parâmetros: `x, y` Dois valores de ponto flutuante
  Retorna: O valor do primeiro parâmetro elevado à potência do segundo parâmetro

- `static long round(double x)`
  Esse método retorna o inteiro `long` mais próximo do parâmetro.
  Parâmetros: `x` Um valor de ponto flutuante
  Retorna: O valor do parâmetro arredondado para o `long` mais próximo

- `static double sin(double radians)`
  Esse método retorna o seno de um dado ângulo em radianos.
  Parâmetros: `radians` Um ângulo, em radianos
  Retorna: Seno do parâmetro

- `static double sqrt(double x)`
  Esse método retorna a raiz quadrada de x, $\sqrt{x}$.
  Parâmetros: `x` Um valor de ponto flutuante não-negativo
  Retorna: Raiz quadrada do parâmetro

- `static double tan(double radians)`
  Esse método retorna a tangente de um dado ângulo em radianos.
  Parâmetros: `radians` Um ângulo, em radianos
  Retorna: Tangente do parâmetro

- `static double toDegrees(double radian)`
  Esse método converte radianos em graus.
  Parâmetros: `radians` Um ângulo, em radianos
  Retorna: Ângulo em graus
- `static double toRadians(double degrees)`
  Esse método converte graus em radianos.
  Parâmetros: `degrees` Um ângulo, em graus
  Retorna: Ângulo em radianos
- `static final double E`
  Essa constante é o valor de *e*, a base dos logaritmos naturais.
- `static final double PI`
  Essa constante é o valor de π

Classe `java.lang.NullPointerException`

Essa exceção é lançada quando um programa tenta utilizar um objeto por meio de uma referência `null`.

Classe `java.lang.NumberFormatException`

Essa exceção é lançada quando um programa tenta analisar sintaticamente o valor numérico de uma string que não é um número.

Classe `java.lang.Object`

- `protected Object clone()`
  Constrói e retorna uma cópia desse objeto cujas variáveis de instância são cópias das variáveis de instância desse objeto. Se uma variável de instância do objeto for a própria referência a um objeto, apenas a referência é copiada, não o objeto. Entretanto, se a classe não implementar a interface `Cloneable`, uma exceção do tipo `CloneNotSupportedException` será lançada. As subclasses devem redefinir esse método para fazer uma cópia profunda.
  Retorna: Uma cópia desse objeto
- `boolean equals(Object other)`
  Esse método testa se `this` e o outro objeto são iguais. Esse método testa somente se as referências a objeto são para o mesmo objeto. As subclasses devem redefinir esse método para comparar as variáveis de instância.
  Parâmetros: `other` Objeto com o qual comparar
  Retorna: `true` se os objetos forem iguais, `false` caso contrário
- `String toString()`
  Esse método retorna uma representação de string desse objeto. Esse método produz somente o nome de classe e as localizações dos objetos. As subclasses devem redefinir esse método para imprimir as variáveis de instância.
  Retorna: String que descreve esse objeto

Classe `java.lang.RuntimeException`

Superclasse para todas as exceções não-verificadas.

Classe `java.lang.String`

- `int compareTo(String other)`
  Esse método compara essa string com a outra string lexicograficamente.
  Parâmetros: `other` Outra string a ser comparada
  Retorna: Um valor menor que 0 se essa string for lexicograficamente menor que a outra, 0 se as strings forem iguais e um valor maior do que 0 caso contrário

- `boolean equals(String other)`
- `boolean equalsIgnoreCase(String other)`
  Esses métodos testam se duas strings são iguais ou se são iguais quando a distinção entre maiúsculas e minúsculas é ignorada.
  Parâmetros: `other` Outra string a ser comparada
  Retorna: `true` se as strings forem iguais

- `static String format(String format, Object... values)`
  Esse método formata a string dada substituindo marcadores de lugar que iniciam com `%` pelos valores dados.
  Parâmetros: `format` String com os marcadores de lugar
  `values` Valores a substituir os marcadores de lugar
  Retorna: A string formatada, com os marcadores de lugar substituídos pelos valores dados

- `int length()`
  Esse método retorna o comprimento dessa string.
  Retorna: Contagem de caracteres nessa string

- `String replace(String match, String replacement)`
  Esse método substitui substrings correspondentes por uma dada string substituta.
  Parâmetros: `match` String cujas correspondências devem ser substituídas
  `replacement` String cujas substrings correspondentes serão substituídas
  Retorna: Uma string idêntica a essa string, com todas as substrings correspondentes substituídas pela string substituta fornecida

- `String substring(int begin)`
- `String substring(int begin, int pastEnd)`
  Esses métodos retornam uma nova string que é uma substring dessa string, composta de todos os caracteres que iniciam na posição `begin` até a posição `pastEnd - 1`, se ela for dada, ou o fim da string.
  Parâmetros: `begin` Índice inicial, inclusive
  `pastEnd` Índice final, exclusive
  Retorna: Substring especificada

- `String toLowerCase()`
  Esse método retorna uma nova string que consiste em todos os caracteres dessa string convertidos em letras minúsculas.
  Retorna: String com todos os caracteres dessa string convertidos em letras minúsculas

- `String toUpperCase()`
  Esse método retorna uma nova string que consiste em todos os caracteres dessa string convertidos em letras maiúsculas.
  Retorna: String com todos os caracteres dessa string convertidos em letras maiúsculas

Classe `java.lang.System`

- `static void arraycopy(`
  `Object from, int fromStart, Object to, int toStart, int count)`
  Esse método copia valores de um array para o outro. (Os parâmetros de array são do tipo `Object` porque você pode converter um array de números em um `Object` mas não em um `Object[]`.)
  Parâmetros: `from` Array de origem
  `fromStart` Posição inicial no array de origem
  `to` Array de destino
  `toStart` Posição inicial nos dados de destino
  `count` Número de elementos do array a ser copiado
- `static long currentTimeMillis()`
  Esse método retorna a diferença, medida em milissegundos, entre a hora atual e a zero hora, hora universal, de 1º de janeiro de 1970.
  Retorna: Hora atual em milissegundos
- `static void exit(int status)`
  Esse método termina o programa.
  Parâmetros: `status` Status de saída. Um código de status diferente de zero indica término anormal
- `static final InputStream in`
  Esse objeto é o fluxo de "entrada padrão". Ler esse fluxo em geral significa ler a entrada do teclado.
- `static final PrintStream out`
  Esse objeto é o fluxo de "saída padrão". Imprimir para esse fluxo em geral envia a saída para a janela de console.

Classe `java.lang.Throwable`

Essa é a superclasse de exceções e erros.

- `Throwable()`
  Constrói um `Throwable` sem mensagem detalhada.
- `String getMessage()`
  Esse método obtém a mensagem que descreve a exceção ou erro.
  Retorna: Mensagem
- `void printStackTrace()`
  Esse método imprime um rastreamento de pilha para o fluxo padrão de "erro". O rastreamento de pilha contém uma impressão desse objeto e de todas as chamadas que estavam pendentes na hora em que ele foi criado.

Pacote java.math

Classe java.math.BigDecimal

- **BigDecimal**(String value)
  Constrói um número de ponto flutuante de precisão arbitrária com os dígitos da string dada.
  Parâmetros: value String que representa o número de ponto flutuante
- BigDecimal **add**(BigDecimal other)
- BigDecimal **multiply**(BigDecimal other)
- BigDecimal **subtract**(BigDecimal other)
  Esses métodos retornam um BigDecimal cujo valor é a soma, a diferença, o produto ou o quociente desse e do outro número.
  Parâmetros: other Outro número
  Retorna: Resultado da operação aritmética

Classe java.math.BigInteger

- **BigInteger**(String value)
  Constrói um inteiro de precisão arbitrária a partir dos dígitos na string dada.
  Parâmetros: value String que representa um inteiro de precisão arbitrária
- BigInteger **add**(BigInteger other)
- BigInteger **divide**(BigInteger other)
- BigInteger **mod**(BigInteger other)
- BigInteger **multiply**(BigInteger other)
- BigInteger **subtract**(BigInteger other)
  Esses métodos retornam um BigInteger cujo valor é a soma, a diferença, o produto, o quociente ou o resto desse e do outro número.
  Parâmetros: other Outro número
  Retorna: Resultado da operação aritmética

Pacote java.util

Classe java.util.ArrayList<E>

- **ArrayList**()
  Constrói uma lista de arrays vazia.
- boolean **add**(E element)
  Esse método acrescenta um elemento ao fim dessa lista de arrays.
  Parâmetros: element Elemento a adicionar
  Retorna: true (Esse método retorna um valor porque sobrescreve um método da interface List.)
- void **add**(int index, E element)
  Esse método insere um elemento nessa lista de arrays.
  Parâmetros: index Posição da inserção
  element Elemento a inserir

- `E get(int index)`
  Esse método obtém o elemento na posição especificada dessa lista de arrays.
  Parâmetros: `index` Posição do elemento a retornar
  Retorna: Elemento solicitado
- `E remove(int index)`
  Esse método remove o elemento na posição especificada dessa lista de arrays e o retorna.
  Parâmetros: `index` Posição do elemento a remover
  Retorna: Elemento removido
- `E set(int index, E element)`
  Esse método substitui o elemento em uma posição especificada dessa lista de arrays.
  Parâmetros: `index` Posição do elemento a substituir
  `element` Elemento a ser armazenado na posição especificada
  Retorna: Elemento que havia antes na posição especificada
- `int size()`
  Esse método retorna o número de elementos dessa lista de arrays.
  Retorna: Número de elementos dessa lista de arrays

Classe `java.util.Arrays`

- `static int binarySearch(Object[] a, Object key)`
  Esse método pesquisa o array especificado em busca do objeto especificado utilizando o algoritmo de pesquisa binária. Os elementos do array devem implementar a interface `Comparable`. O array deve ser classificado em ordem crescente.
  Parâmetros: `a` Array a ser pesquisado
  `key` Valor a ser procurado
  Retorna: Posição da chave de pesquisa, se ela estiver contida no array; caso contrário,– *índice* – 1, onde *índice* é a posição onde o elemento pode ser inserido
- `static void sort(Object[] a)`
  Esse método classifica o array especificado de objetos em ordem crescente. Seus elementos devem implementar a interface `Comparable`.
  Parâmetros: `a` Array a ser classificado

Classe `java.util.Calendar`

- `int get(int field)`
  Esse método retorna o valor do campo dado.
  Parâmetros: Um entre `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.DAY_OF_MONTH`, `Calendar.HOUR`, `Calendar.MINUTE`, `Calendar.SECOND` e `Calendar.MILLISECOND`

Interface `java.util.Collection<E>`

- `boolean add(E element)`
  Esse método adiciona um elemento a essa coleção.
  Parâmetros: `element` Elemento a adicionar
  Retorna: `true` se a adição do elemento alterar a coleção

- `boolean contains(E element)`
  Esse método testa se um elemento está presente nessa coleção.
  Parâmetros: `element` Elemento a localizar
  Retorna: `true` se o elemento estiver contido na coleção
- `Iterator iterator()`
  Esse método retorna um iterador que pode ser utilizado para percorrer os elementos dessa coleção.
  Retorna: Objeto de uma classe que implementa a interface `Iterator`
- `boolean remove(E element)`
  Esse método remove um elemento dessa coleção.
  Parâmetros: `element` Elemento a remover
  Retorna: `true` se a remoção do elemento alterar a coleção
- `int size()`
  Esse método retorna o número de elementos nessa coleção.
  Retorna: Número de elementos nessa coleção

Classe `java.util.Collections`

- `static <T> int binarySearch(List<T> a, T key)`
  Esse método pesquisa na lista especificada o objeto especificado utilizando o algoritmo de pesquisa binária. Os elementos da lista devem implementar a interface `Comparable`. A lista deve estar classificada em ordem crescente.
  Parâmetros: `a` Lista a ser pesquisada
  `key` Valor a ser procurado
  Retorna: Posição da chave de pesquisa, se ela estiver contida na lista; caso contrário,– *índice* – 1, onde *índice* é a posição onde o elemento pode ser inserido
- `static <T> void sort(T[] a)`
  Esse método classifica a lista especificada de objetos em ordem crescente. Seus elementos devem implementar a interface `Comparable`.
  Parâmetros: `a` Lista a ser classificada

Interface `java.util.Comparator<T>`

- `int compare(T first, T second)`
  Esse método compara os objetos dados.
  Parâmetros: `first`, `second` Objetos a serem comparados
  Retorna: Um inteiro negativo se o primeiro objeto for menor que o segundo, zero se forem iguais ou um inteiro positivo caso contrário

Classe `java.util.EventObject`

- `Object getSource()`
  Esse método retorna uma referência ao objeto onde esse evento ocorreu inicialmente.
  Retorna: A origem desse evento

Classe `java.util.GregorianCalendar`

- `GregorianCalendar()`
  Constrói um objeto calendário que representa a data e a hora atuais.

- `GregorianCalendar(int year, int month, int day)`
  Constrói um objeto calendário que representa o início da data dada.
  Parâmetros: `year`, `month`, `day` Data fornecida

Classe `java.util.HashMap<K, V>`

- `HashMap<K, V>()`
  Constrói um mapa de hash vazio.

Classe `java.util.HashSet<K, V>`

- `HashSet<K, V>()`
  Constrói um conjunto de hash vazio.

Classe `java.util.InputMismatchException`

Essa exceção é lançada se o próximo item de entrada disponível não corresponder ao tipo do item solicitado.

Interface `java.util.Iterator<E>`

- `boolean hasNext()`
  Esse método verifica se o iterador ultrapassou o fim da lista.
  Retorna: `true` se o iterador ainda não ultrapassou o fim da lista
- `E next()`
  Esse método move o iterador para o próximo elemento da lista encadeada. Ele lança uma exceção se o iterador ultrapassou o fim da lista.
  Retorna: Objeto que acabou de ser pulado
- `void remove()`
  Esse método remove o elemento que foi retornado pela última chamada a `next` ou `previous`. Lança uma exceção se houver uma operação `add` ou `remove` depois da última chamada a `next` ou `previous`.

Classe `java.util.LinkedList<E>`

- `void addFirst(E element)`
- `void addLast(E element)`
  Esses métodos adicionam um elemento antes do primeiro ou depois do último elemento nessa lista.
  Parâmetros: `element` Elemento a ser adicionado
- `E getFirst()`
- `E getLast()`
  Esses métodos retornam uma referência ao elemento especificado dessa lista.
  Retorna: Primeiro ou último elemento
- `E removeFirst()`
- `E removeLast()`
  Esses métodos removem o elemento especificado dessa lista.
  Retorna: Uma referência ao elemento removido

Interface `java.util.List<E>`

- `ListIterator<E> listIterator()`
  Esse método obtém um iterador para visitar os elementos nessa lista.
  Retorna: Iterador que aponta para antes do primeiro elemento dessa lista

Interface `java.util.ListIterator<E>`

Os objetos que implementam essa interface são criados pelos métodos `listIterator` das classes de lista.

- `void add(E element)`
  Esse método adiciona um elemento depois da posição do iterador e move o iterador para depois do novo elemento.
  Parâmetros: `element` Elemento a ser adicionado
- `boolean hasPrevious()`
  Esse método verifica se o iterador está antes do primeiro elemento da lista.
  Retorna: `true` se o iterador não estiver antes do primeiro elemento da lista
- `E previous()`
  Esse método move o iterador para o elemento anterior na lista encadeada. Esse método lança uma exceção se o iterador estiver antes do primeiro elemento da lista.
  Retorna: Objeto que acabou de ser pulado
- `void set(E element)`
  Esse método substitui o elemento que foi retornado pela última chamada a `next` ou `previous`. Esse método lança uma exceção se houver uma operação de adição ou remoção depois da última chamada a `next` ou `previous`.
  Parâmetros: `element` Elemento que substitui o antigo elemento da lista

Interface `java.util.Map<K, V>`

- `V get(K key)`
  Obtém o valor associado com uma chave nesse mapa.
  Parâmetros: `key` Chave através da qual localizar o valor associado
  Retorna: Valor associado com a chave, ou `null` se a chave não estiver presente na tabela
- `Set<K> keySet()`
  Esse método retorna todas as chaves na tabela desse mapa.
  Retorna: Conjunto de todas as chaves na tabela desse mapa
- `V put(K key, V value)`
  Esse método associa um valor com uma chave nesse mapa.
  Parâmetros: `key` Chave de pesquisa
  `value` Valor associado com a chave
  Retorna: Valor anteriormente associado com a chave ou `null` se a chave não estiver presente na tabela
- `V remove(K key)`
  Esse método remove uma chave e seu valor associado desse mapa.
  Parâmetros: `key` Chave de pesquisa
  Retorna: O valor anteriormente associado com a chave ou `null` se a chave não estiver presente na tabela

Classe `java.util.NoSuchElementException`

Essa exceção é lançada se for feita uma tentativa de recuperar um valor inexistente.

Classe `java.util.PriorityQueue<E>`

- `PriorityQueue<E>()`
  Constrói uma fila de prioridade vazia. O tipo de elemento `E` deve implementar a interface `Comparable`.
- `E remove()`
  Esse método remove o menor elemento na fila de prioridade.
  Retorna: Valor removido

Classe `java.util.Random`

- `Random()`
  Constrói um novo gerador de número aleatório.
- `double nextDouble()`
  Esse método retorna o próximo número de ponto flutuante pseudoaleatório, uniformemente distribuído entre 0.0 (inclusivo) e 1.0 (exclusivo) da seqüência desse gerador de números aleatórios.
  Retorna: O próximo número de ponto flutuante pseudoaleatório
- `int nextInt(int n)`
  Esse método retorna o próximo inteiro pseudoaleatório, uniformemente distribuído entre o valor 0 (inclusive) e o valor especificado (exclusive) elaborado a partir da seqüência desse gerador de números aleatórios.
  Parâmetros: `n` Número de valores a gerar
  Retorna: Próximo inteiro pseudoaleatório

Classe `java.util.Scanner`

- `Scanner(InputStream in)`
- `Scanner(Reader in)`
  Esses métodos constroem um scanner que lê a partir de um dado fluxo de entrada ou leitor.
  Parâmetros: `in` Fluxo de entrada ou leitor a partir do qual fazer a leitura
- `void close()`
  Esse método fecha esse scanner e libera todos os recursos de sistema associados.
- `boolean hasNext()`
- `boolean hasNextDouble()`
- `boolean hasNextInt()`
- `boolean hasNextLine()`
  Esses métodos testam se é possível ler qualquer string não-vazia, um valor de ponto flutuante, um inteiro ou uma linha, como o próximo item.
  Retorna: `true` se for possível ler um item do tipo solicitado, `false` caso contrário (seja porque o fim do arquivo foi alcançado ou porque um tipo de número foi testado e o próximo item não é um número)

- `String next()`
- `double nextDouble()`
- `int nextInt()`
- `String nextLine()`
  Esses métodos lêem a próxima string delimitada por espaços em branco, valor de ponto flutuante, inteiro ou linha.
  Retorna: A string lida

Interface `java.util.Set<E>`

Essa interface descreve uma coleção que contém elementos não-duplicados.

Classe `java.util.TreeMap<K, V>`

- `TreeMap<K, V>()`
  Constrói um mapa de árvore vazio.

Classe `java.util.TreeSet<K, V>`

- `TreeSet<K, V>()`
  Constrói um conjunto de árvore vazio.

Pacote `java.util.logging`

Classe `java.util.logging.Level`

- `static final int ALL`
  Esse valor indica o registro de log de todas as mensagens.
- `static final int INFO`
  Esse valor indica o registro de log informacional.
- `static final int NONE`
  Esse valor indica o registro de log de nenhuma mensagem.

Classe `java.util.logging.Logger`

- `static Logger getLogger(String id)`
  Esse método obtém o registro de um dado ID. Utilize o ID `"global"` para obter o registro global padrão.
  Parâmetros: `id` ID de registro como `"global"` ou `"com.mycompany.mymodule"`
  Retorna: Registro com o dado ID
- `void info(String message)`
  Esse método registra em log uma mensagem informacional.
  Parâmetros: `message` Mensagem a registrar em log
- `void setLevel(Level aLevel)`
  Esse método configura o nível de registro em log. As mensagens registradas em log com menos rigor do que o nível atual são ignoradas.
  Parâmetros: `aLevel` Nível mínimo para mensagens registradas em log

Pacote javax.swing

Classe javax.swing.AbstractButton

- `void addActionListener(ActionListener listener)`
  Esse método adiciona um ouvinte de ação ao botão.
  Parâmetros: `listener` Ouvinte de ação a ser adicionado
- `boolean isSelected()`
  Esse método retorna o estado de seleção do botão.
  Retorna: `true` se o botão for selecionado
- `void setSelected(boolean state)`
  Esse método configura o estado de seleção do botão. Esse método atualiza o botão, mas não desencadeia um evento de ação.
  Parâmetros: `state` `true` para selecionar, `false` para remover a seleção

Classe javax.swing.ButtonGroup

- `void add(AbstractButton button)`
  Esse método adiciona o botão ao grupo.
  Parâmetros: `button` Botão a adicionar

Classe javax.swing.ImageIcon

- `ImageIcon(String filename)`
  Constrói um ícone de imagem com o arquivo gráfico especificado.
  Parâmetros: `filename` String que especifica um nome de arquivo

Classe javax.swing.JButton

- `JButton(String label)`
  Constrói um botão com o rótulo dado.
  Parâmetros: `label` Rótulo do botão

Classe javax.swing.JCheckBox

- `JCheckBox(String text)`
  Constrói uma caixa de seleção contendo o texto dado e inicialmente desmarcada. (Utilize o método `setSelected()` para criar a caixa selecionada; ver a classe `javax.swing.AbstractButton`.)
  Parâmetros: `text` Texto exibido perto da caixa de seleção

Classe javax.swing.JComboBox

- `JComboBox()`
  Constrói uma caixa de combinação sem itens.
- `void addItem(Object item)`
  Esse método adiciona um item à lista de itens dessa caixa de combinação.
  Parâmetros: `item` Item a adicionar

- `Object` **`getSelectedItem()`**
  Esse método obtém o item atualmente selecionado dessa caixa de combinação.
  Retorna: Item atualmente selecionado

- `boolean` **`isEditable()`**
  Esse método verifica se a caixa de combinação é editável. Uma caixa de combinação editável permite ao usuário digitar no campo de texto da caixa de combinação.
  Retorna: `true` se a caixa de combinação for editável

- `void` **`setEditable`**`(boolean state)`
  Esse método é utilizado para tornar a caixa de combinação editável ou não.
  Parâmetros: `state` `true` para tornar editável, `false` para desativar a edição

Classe `javax.swing.JComponent`

- `protected void` **`paintComponent`**`(Graphics g)`
  Sobrescreva esse método para desenhar a superfície de um componente. Seu método precisa chamar `super.paintComponent(g)`.
  Parâmetros: `g` Contexto de elementos gráficos utilizado para desenhar

- `void` **`setBorder`**`(Border b)`
  Esse método configura a borda desse componente.
  Parâmetros: `b` Borda para envolver esse componente

- `void` **`setFont`**`(Font f)`
  Configura a fonte utilizada para o texto nesse componente.
  Parâmetros: `f` Fonte

Classe `javax.swing.JFileChooser`

- **`JFileChooser()`**
  Constrói um seletor de arquivo.

- `File` **`getSelectedFile()`**
  Esse método obtém o arquivo selecionado desse seletor de arquivo.
  Retorna: Arquivo selecionado

- `int` **`showOpenDialog`**`(Component parent)`
  Esse método exibe uma caixa de diálogo de seleção de arquivo "Open File".
  Parâmetros: `parent` Componente pai ou `null`
  Retorna: Estado de retorno desse seletor de arquivo depois de ele ter sido fechado pelo usuário: `APPROVE_OPTION` ou `CANCEL_OPTION`. Se o estado `APPROVE_OPTION` for retornado, chame `getSelectedFile()` nesse seletor de arquivo para obter o arquivo

- `int` **`showSaveDialog`**`(Component parent)`
  Esse método exibe uma caixa de diálogo de seleção de arquivo "Save File".
  Parâmetros: `parent` Componente pai ou `null`
  Retorna: Estado de retorno do seletor de arquivo depois de ele ter sido fechado pelo usuário: `APPROVE_OPTION` ou `CANCEL_OPTION`

Classe `javax.swing.JFrame`

- `void setDefaultCloseOperation(int operation)`
  Esse método configura a ação padrão para fechar o frame.
  Parâmetros: `operation` Operação de fechar desejada. Escolha entre `DO_NOTHING_ON_CLOSE`, `HIDE_ON_CLOSE` (o padrão), `DISPOSE_ON_CLOSE` ou `EXIT_ON_CLOSE`
- `void setJMenuBar(JMenuBar mb)`
  Esse método configura a barra de menus para esse frame.
  Parâmetros: `mb` Barra de menus. Se `mb` for `null`, então a barra de menu atual é removida
- `static final int EXIT_ON_CLOSE`
  Esse valor indica que quando o usuário fechar esse frame, o aplicativo deve encerrar sua execução.

Classe `javax.swing.JLabel`

- `JLabel(String text)`
- `JLabel(String text, int alignment)`
  Esses contêineres criam uma instância `JLabel` com o texto especificado e o alinhamento horizontal.
  Parâmetros: `text` Texto de rótulo deve ser exibido no rótulo
  `alignment` Um entre `SwingConstants.LEFT`, `SwingConstants.CENTER` e `SwingConstants.RIGHT`

Classe `javax.swing.JMenu`

- `JMenu()`
  Constrói um menu sem itens.
- `JMenuItem add(JMenuItem menuItem)`
  Esse método acrescenta um item de menu ao fim desse menu.
  Parâmetros: `menuItem` Item de menu a ser adicionado
  Retorna: Item de menu que foi adicionado

Classe `javax.swing.JMenuBar`

- `JMenuBar()`
  Constrói uma barra de menus sem menus.
- `JMenu add(JMenu menu)`
  Esse método acrescenta um menu ao fim dessa barra de menus.
  Parâmetros: `menu` Menu a ser adicionado
  Retorna: Menu que foi adicionado

Classe `javax.swing.JMenuItem`

- `JMenuItem(String text)`
  Constrói um item de menu.
  Parâmetros: `text` Texto a ser exibido no item de menu

Classe `javax.swing.JOptionPane`

- `static String showInputDialog(Object prompt)`
  Esse método abre uma caixa de diálogo de entrada modal, que exibe um prompt e espera o usuário inserir uma entrada em um campo de texto, impedindo que o usuário faça qualquer outra coisa nesse programa.
  Parâmetros: `prompt` O prompt a ser exibido
  Retorna: String que o usuário digitou

- `static void showMessageDialog(Component parent, Object message)`
  Esse método abre uma caixa de diálogo de confirmação que exibe uma mensagem e espera o usuário confirmá-la.
  Parâmetros: `parent` Componente pai ou `null`
  `message` Mensagem a exibir

Classe `javax.swing.JPanel`

Essa classe é um componente sem decorações. Ela pode ser utilizada como um contêiner invisível para outros componentes.

Classe `javax.swing.JRadioButton`

- `JRadioButton(String text)`
  Constrói um botão de opção com o texto dado que inicialmente está não-selecionado. (Utilize o método `setSelected()` para selecioná-lo; veja a classe `javax.swing.AbstractButton`.)
  Parâmetros: `text` String exibida ao lado do botão de opção

Classe `javax.swing.JScrollPane`

- `JScrollPane(Component c)`
  Constrói um painel de rolagem em torno do componente dado.
  Parâmetros: `c` Componente é decorado com barras de rolagens

Classe `javax.swing.JSlider`

- `JSlider(int min, int max, int value)`
  Esse construtor cria um controle deslizante horizontal que usa o mínimo, o máximo e o valor dados.
  Parâmetros: `min` Menor valor de controle deslizante possível
  `max` Maior valor de controle deslizante possível
  `value` Valor inicial do controle deslizante

- `void addChangeListener(ChangeListener listener)`
  Esse método adiciona um ouvinte de alteração ao controle deslizante.
  Parâmetros: `listener` Ouvinte de alteração a adicionar

- `int getValue()`
  Esse método retorna o valor do controle deslizante.
  Retorna: Valor atual do controle deslizante

Classe `javax.swing.JTextArea`

- `JTextArea()`
  Constrói uma área de texto vazia.
- `JTextArea(int rows, int columns)`
  Constrói uma área de texto vazia com o número especificado de linhas e colunas.
  Parâmetros: `rows` Número de linhas
  `columns` Número de colunas
- `void append(String text)`
  Esse método acrescenta texto a essa área de texto.
  Parâmetros: `text` Texto a acrescentar

Classe `javax.swing.JTextField`

- `JTextField()`
  Constrói um campo de texto vazio.
- `JTextField(int columns)`
  Constrói um campo de texto vazio com o número especificado de colunas.
  Parâmetros: `columns` Número de colunas

Classe `javax.swing.Timer`

- `Timer(int millis, ActionListener listener)`
  Constrói um timer que notifica um ouvinte de ação sempre que um intervalo de tempo decorrer.
  Parâmetros: `millis` Número de milissegundos entre notificações de timer
  `listener` Objeto a ser notificado quando transcorrer o intervalo de tempo
- `void start()`
  Esse método inicia o timer. Uma vez que o timer iniciou, ele começa a notificar seu ouvinte.
- `void stop()`
  Esse método pára o timer. Uma vez que o timer parou, ele não notifica mais seu ouvinte.

Pacote `javax.swing.border`

Classe `javax.swing.border.EtchedBorder`

- `EtchedBorder()`
  Esse construtor cria uma borda inferior chanfrada.

Classe `javax.swing.border.TitledBorder`

- `TitledBorder(Border b, String title)`
  Esse construtor cria uma borda intitulada que adiciona um título a uma borda dada.
  Parâmetros: `b` Borda à qual o título é adicionado
  `title` Título que a borda deve exibir

Pacote `javax.swing.event`

Classe `javax.swing.event.ChangeEvent`

Componentes como controles deslizantes emitem eventos de alteração quando são manipulados pelo usuário.

Interface `javax.swing.event.ChangeListener`

- `void stateChanged(ChangeEvent e)`
  Esse evento é chamado quando a fonte do evento mudou seu estado.
  Parâmetros: `e` Evento de alteração

Pacote `javax.swing.text`

Classe `javax.swing.text.JTextComponent`

- `String getText()`
  Esse método retorna o texto contido nesse componente de texto.
  Retorna: Texto
- `boolean isEditable()`
  Esse método verifica se esse componente de texto é editável.
  Retorna: `true` se o componente for editável
- `void setEditable(boolean state)`
  Esse método é utilizado para tornar esse componente de texto editável ou não.
  Parâmetros: `state` `true` para tornar editável, `false` para desativar a edição
- `void setText(String text)`
  Esse método configura o texto desse componente de texto com o texto especificado. Se o texto estiver vazio, o antigo texto é excluído.
  Parâmetros: `text` Novo texto a ser configurado

Glossário

Abrir um arquivo Preparar um arquivo para leitura ou gravação.

Abstração Processo de localizar o conjunto de recursos essenciais de um bloco de construção de um programa, como uma classe.

Acesso aleatório Capacidade de acessar qualquer valor diretamente, sem ler os valores que o precedem.

Acesso de pacote Acessibilidade por métodos de classes no mesmo pacote.

Acesso seqüencial Acessar valores um depois do outro sem pular nenhum deles.

Acoplamento Grau em que as classes estão relacionadas entre si por dependência.

Adaptador de evento Classe que implementa uma interface de ouvinte de evento definindo todos os métodos para que não façam nada.

Agregação O relacionamento "*tem um*" entre classes.

Algoritmo Especificação não-ambígua, executável e com condição de término de uma maneira de resolver um problema.

Algoritmo de heapsort Algoritmo de classificação que insere os valores a serem classificados em um heap.

API (Application Programming Interface) Biblioteca de funções para construir programas.

Applet Programa gráfico em Java que executa dentro de um navegador Web ou visualizador de applet.

Argumento Parâmetro real em uma chamada de método ou um dos valores combinado por um operador.

Arquivo binário Arquivo em que os valores são armazenados em sua representação binária e não pode ser lido como texto.

Arquivo de texto Arquivo em que valores são armazenados em sua representação textual.

Arquivo-fonte Arquivo contendo instruções em uma linguagem de programação como Java.

Array Coleção de valores do mesmo tipo armazenada em posições de memória contíguas, cada uma das quais pode ser acessada por um índice do tipo inteiro.

Array abstrato Seqüência ordenada de itens que pode ser acessada aleatoriamente e de modo eficiente por um índice do tipo inteiro.

Array bidimensional Arranjo tabular de elementos onde um elemento é especificado por um índice de linha e um índice de coluna.

Array parcialmente preenchido Array que não é preenchido até a capacidade máxima, acompanhado de uma variável indicando o número de elementos realmente armazenados.

Arrays paralelos As arrays de mesmo comprimento, em que elementos correspondentes são relacionados logicamente.

Árvore Estrutura de dados que consiste em nós, cada um dos quais com uma lista de nós filhos. Um dos nós é distinguido como o nó raiz.

Árvore binária Árvore em que cada nó tem no máximo dois nós filhos.

Árvore de pesquisa binária Árvore binária em que *cada* subárvore tem a propriedade de que todos os descendentes à esquerda sejam menores que o valor armazenado na raiz e todos os descendentes à direita sejam maiores.

Árvore equilibrada Árvore onde cada subárvore tem a propriedade de que o número de descendentes à esquerda seja quase o mesmo que o número de descendentes à direita.

Assertiva Afirmação de que certa condição é mantida em uma determinada localização do programa.

Assinatura de método Nome de um método e os tipos de seus parâmetros.

Associação Relacionamento entre classes em que se pode navegar de objetos de uma classe para objetos da outra classe, normalmente seguindo referências a objeto.

Associatividade dos operadores Regra que determina em que ordem operadores da mesma precedência são executados. Por exemplo, em Java, a associatividade do operador `-` é da esquerda para a direita porque `a - b - c` é interpretado como `(a - b) - c` e a associatividade `=` é da direita para a esquerda porque `a = b = c` é interpretado como `a = (b = c)`.

Atribuição Colocar um novo valor em uma variável.

Atributo Propriedade identificada que um objeto deve manter.

Auto-boxing Converte automaticamente um valor de tipo primitivo em um objeto do tipo empacotador (*wrapper*).

Avaliação de curto-circuito Avaliar somente uma parte de uma expressão quando o restante não altera o resultado.

Avaliação preguiçosa Adiar o cálculo de um valor até que ele seja requerido, evitando o cálculo se o valor nunca for necessário.

Biblioteca Conjunto de classes precompiladas que pode ser incluído em programas.

Bit Dígito binário; a menor unidade de informações, tendo dois possíveis valores: 0 e 1. Um elemento de dados consistindo em n bits aceita 2^n valores possíveis.

Bit de sinal Bit de um número binário que indica se o número é positivo ou negativo.

Bloco Um grupo de instruções entre chaves `{}`.

Bloco aninhado Bloco contido em outro bloco.

Botão de opção Componente de interface com o usuário que pode ser utilizado para selecionar uma de várias opções.

break, comando Comando que termina um laço ou um comando `switch`.

Bucket Em uma tabela hash, um conjunto de valores com o mesmo código de hash.

Buffer Local de armazenamento temporário para guardar valores que foram produzidos (por exemplo, caracteres digitados pelo usuário) e estão esperando para ser consumidos (por exemplo, ler uma linha por vez).

Bug Erro de programação.

Byte Número composto de oito bits. Atualmente todos os computadores fabricados utilizam o byte como a menor unidade de armazenamento em memória.

Bytecode Instruções para a máquina virtual Java.

Caixa de combinação Componente de interface com o usuário que combina um campo de texto com uma lista suspensa de seleções.

Caixa de seleção Componente de interface com o usuário que pode ser utilizado para uma seleção binária.

Caminho de classe Conjunto de diretórios e repositórios de arquivo no qual a máquina virtual procura por arquivos de classe.

Campo de instância Variável definida em uma classe onde cada objeto da classe tem seu próprio valor.

Campo de texto Componente de interface com o usuário que permite que um usuário forneça entrada de texto.

Campo estático Variável definida em uma classe que tem apenas um valor para a classe inteira, que pode ser acessada e alterada por qualquer método dessa classe.

Caractere Uma única letra, dígito ou símbolo.

Caractere de escape Caractere no texto que não é considerado literalmente, mas tem um significado especial quando combinado com o caractere ou caracteres que o seguem. O caractere `\` é o caractere de escape em strings Java.

Caractere de tabulação Caractere `'\t'`, que avança o cursor para a próxima posição de um conjunto de posições fixas conhecidas como paradas de tabulação.

Caso de teste de limite Caso de teste envolvendo valores que estão no limite externo do conjunto de valores válidos. Por exemplo, se uma função deve funcionar para todos os inteiros não-negativos, então 0 é um caso de teste de limite.

Caso de teste negativo Caso de teste cuja falha é esperada. Por exemplo, ao testar um programa de cálculo de raiz, uma tentativa de calcular a raiz quadrada de −1 é um caso de teste negativo.

Caso de teste positivo Caso de teste que se espera que o método trate corretamente.

`catch`, cláusula Parte de um bloco `try` que é executada quando uma exceção correspondente é lançada por qualquer instrução no bloco `try`.

Chamada por referência Mecanismo de chamada de método em que o método recebe a localização na memória de uma variável fornecida como um parâmetro real. A chamada por referência permite que um método altere o conteúdo da variável original de modo que o efeito da alteração permaneça depois de o método retornar.

Chamada por valor Mecanismo de chamada de método em que o método recebe uma cópia do conteúdo de uma variável fornecida como um parâmetro real. Java usa somente chamadas por valor. Se o tipo de uma variável de parâmetro for uma classe, seu valor é uma referência a um objeto, portanto, o método pode alterar esse objeto, mas não pode fazer a variável de parâmetro referenciar um objeto diferente.

Ciclo de vida do software Todas as atividades relacionadas à criação e manutenção do software, da análise inicial até a obsolescência.

Classe Tipo de dados definido pelo programador.

Classe abstrata Classe que não pode ser instanciada.

Classe anônima Classe que não tem um nome.

Classe concreta Classe que pode ser instanciada.

Classe de evento Classe que contém informações sobre um evento, como sua fonte.

Classe empacotadora (Wrapper) Classe que contém um valor de tipo primitivo, como `Integer`.

Classe genérica Classe com um ou mais parâmetros de tipo.

Classe imutável Classe sem métodos modificadores.

Classe interna Classe que é definida dentro de outra classe.

Classificação por intercalação Algoritmo de classificação que primeiro classifica duas metades de uma estrutura de dados e, então, intercala os subarrays classificados.

Classificação por seleção Algoritmo de classificação onde o menor elemento é repetidamente localizado e removido até que não reste nenhum elemento.

Clonagem Fazer uma cópia de um objeto cujo estado pode ser modificado independentemente do objeto original.

Cobertura do teste Instruções de um programa que são executadas em um conjunto de casos de teste.

Código de hash Valor que é calculado por uma função de hash.

Código de máquina Instruções que podem ser executadas diretamente pela CPU.

Código espaguete Fluxo de controle complicado e difícil de entender.

Código legado Software que existe há muito tempo e que continua a operar.

Código-fonte Instruções em uma linguagem de programação que precisam ser traduzidas antes da execução em um computador.

Coesão Uma classe é coesa se seus recursos suportam uma abstração simples.

Coerção (typecasting) Conversão explícita de um valor de um tipo em um tipo diferente. Por exemplo, a coerção de um número de ponto flutuante `x` em um inteiro é expressa em Java pela notação de coerção `(int) x`.

Colaboradora Classe da qual outra classe depende.

Coleta de lixo Reivindicação automática da memória que está ocupada por objetos que não são mais referenciados.

Colisão de hash Dois objetos diferentes para os quais uma função de hash calcula valores idênticos.

Comando composto Comando como `if` ou `while` que é composto de várias partes como uma condição e um corpo.

Comentário Uma explicação para ajudar o leitor humano a entender uma seção de um programa; ignorado pelo compilador.

Comentário de documentação Comentário em um arquivo-fonte que pode ser automaticamente extraído do programa por um programa chamado `javadoc`.

Compilador Programa que traduz código escrito em uma linguagem de alto nível (como Java) para instruções de máquina (como bytecodes para a máquina virtual Java).

Componente da interface com o usuário Bloco de construção para uma interface gráfica com o usuário, como um botão ou campo de texto. Os componentes da interface com o usuário são utilizados para apresentar as informações para o usuário e permitem que o usuário insira informações para o programa.

Concatenação Colocar uma string depois da outra para formar uma nova string.

Conflito de nomes Uso acidental do mesmo nome para indicar dois recursos de programa, de modo que não podem ser resolvidos pelo compilador.

Conjunto de testes Conjunto de casos de teste para um programa.

Constante Valor que não pode ser alterado por um programa. Em Java, constantes são definidas com a palavra-chave `final`.

Construção Configurar o estado inicial de um objeto recém-alocado.

Construtor Método que inicializa um objeto recém-instanciado.

Construtor padrão Construtor invocado sem parâmetros.

Contêiner Componente de interface com o usuário que pode armazenar outros componentes e apresentá-los em conjunto para o usuário. Além disso, uma estrutura de dados, como uma lista, que pode armazenar uma coleção de objetos e apresentá-los individualmente para um programa.

Contexto gráfico Classe por meio da qual um programador pode fazer com que formas apareçam em uma janela ou em um mapa de bits não visível.

Cópia superficial Copiar apenas a referência a um objeto.

CPU (ou UCP – Unidade Central de Processamento) Parte de um computador que executa as instruções de máquina.

Dependência Relacionamento *usa* entre classes, em que uma classe precisa de serviços fornecidos por outra classe.

Depurador Programa que permite a um usuário executar outro programa, um ou alguns passos por vez, parar a execução e inspecionar as variáveis a fim de analisar erros.

Diagrama de estado Diagrama que retrata transições de estado e suas causas.

Diagrama de sintaxe Representação gráfica de regras gramaticais.

Diretório Estrutura em um disco que pode armazenar arquivos ou outros diretórios; também chamada de pasta.

Distinção entre letras maiúsculas e minúsculas (Case sensitive) Diferenciação entre letras maiúsculas e minúsculas.

Divisão de inteiros Considerar o quociente de dois inteiros e descartar o resto. Em Java o símbolo `/` indica divisão de inteiro se ambos os argumentos forem inteiros. Por exemplo, `11/4` é `2` e não `2,75`.

Editor Programa para gravar e modificar arquivos de texto.

Efeito colateral Efeito de um método que não corresponde a retornar um valor.

Encapsulamento Ocultamento dos detalhes de implementação.

Entrada armazenada em buffer Entrada agrupada em lotes, por exemplo, uma linha por vez.

Erro de arredondamento Erro introduzido pelo fato de que o computador pode armazenar apenas um número finito de dígitos de um número de ponto flutuante.

Erro de limite Tentar acessar um elemento do array que está fora do intervalo válido.

Erro de lógica Erro em um programa sintaticamente correto que faz com que ele atue diferentemente de sua especificação.

Erro de sintaxe Uma instrução que não segue as regras de programação de linguagem e é rejeitada pelo compilador.

Erro em tempo de compilação Erro que é detectado quando um programa é compilado.

Erro em tempo de execução *Ver* **Erro de lógica**

Erro "por um" Erro de programação comum em que um valor é "um" maior ou menor do que deveria ser.

Escopo Parte de um programa em que uma variável é definida.

Espaço em branco Qualquer seqüência formada de somente por espaços, tabulações e caracteres de nova linha.

Especificador de acesso Palavra-chave que indica a acessibilidade de um recurso, como `private` ou `public`.

Estado Valor atual de um objeto, que é determinado pela ação cumulativa de todos os métodos que foram invocados a partir dele.

Esvaziar um fluxo Enviar todos os caracteres que ainda estão armazenados em um buffer para seu destino.

Exceção Classe que sinaliza uma condição que impede o programa de continuar normalmente. Quando tal condição ocorre, um objeto da classe de exceção é lançado.

Exceção não-verificada Exceção que o compilador não verifica.

Exceção verificada Exceção que o compilador verifica. Todas as exceções verificadas devem ser declaradas ou capturadas.

Explorador de teste Programa que chama uma função que precisa ser testada, fornecendo parâmetros e analisando o valor de retorno da função.

Expressão Construção sintática composta de variáveis, constantes, chamadas de método e operadores que os combinam.

Expressão regular String que define um conjunto de strings correspondentes de acordo com seu conteúdo. Cada parte de uma expressão regular pode ser um caractere requerido específico; um de um conjunto de caracteres permitidos como `[abc]`, que pode ser um intervalo como `[a-z]`; qualquer caractere que não esteja em um conjunto de caracteres proibidos, como `[^0-9]`; uma repetição de uma ou mais correspondências, como `[0-9]+` ou zero ou mais, como `[ACGT]*`; um entre um conjunto de alternativas, como `and|et|und`; ou várias outras possibilidades. Por exemplo, "`[A-Za-z]*[0-9]+`" corresponde a "`Cloud9`" ou "`007`" mas não a "`Jack`".

Extensão Última parte de um nome de arquivo, que especifica o tipo do arquivo. Por exemplo, a extensão `.java` indica um arquivo Java.

Fibonacci, números de Seqüência de números 1, 1, 2, 3, 5, 8, 13, . . ., em que cada termo é a soma de seus dois predecessores.

Ficha CRC Ficha representando uma classe e listando suas responsabilidades e classes colaboradoras.

Fila Coleção de itens recuperados com base na regra "primeiro a entrar, primeiro a sair".

Fila de prioridade Tipo abstrato de dados que permite inserção eficiente de elementos e remoção eficiente do menor elemento.

File Seqüência de bytes que é armazenada em disco.

Fim de arquivo Condição que é verdadeira quando todos os caracteres de um arquivo foram lidos. Observe que não há "caractere de fim de arquivo" especial. Ao compor um arquivo no teclado, você pode precisar digitar um caractere especial para dizer ao sistema operacional que deseja terminar o arquivo, mas esse caractere não faz parte do arquivo.

`finally`, cláusula Parte de um bloco `try` que é executada independentemente de como se dá a saída do bloco `try`.

Flag *Ver* **Tipo booleano**

Fluxo Uma abstração de uma seqüência de bytes a partir da qual dados podem ser lidos ou na qual dados podem ser gravados.

Fonte Conjunto de formas de caracteres com estilo e tamanho específicos.

Fonte do evento Objeto que pode notificar outras classes de eventos.

Frame Janela com uma borda e uma barra de título.

Função de hash Função que calcula um valor do tipo inteiro a partir de um objeto de maneira que diferentes objetos provavelmente resultem em valores diferentes.

Gerenciador de leiaute Classe que organiza componentes de interface com o usuário dentro de um contêiner.

`goto`, comando Comando que transfere o controle para alguma outra instrução, que é marcada com um rótulo. Java não tem comando `goto`.

Gravador Na biblioteca Java de entrada/saída, uma classe para a qual caracteres serão enviados.

`grep` ("global regular expression print") Programa de pesquisa, útil para localizar todas as strings que correspondem a um determinado padrão em um conjunto de arquivos.

GUI (graphical user interface) Interface com o usuário na qual o usuário fornece entradas por meio de componentes gráficos como botões, menus e campos de texto.

Hashing Aplicar uma função de hash a um conjunto de objetos.

Heap Árvore binária equilibrada utilizada para implementar algoritmos de classificação e filas de prioridade.

Herança Relacionamento "*é um*" entre uma superclasse mais geral e uma subclasse mais especializada.

HTML (Hypertext Markup Language) Linguagem em que páginas Web são descritas.

IDE (Ambiente Integrado de Desenvolvimento) Ambiente de programação que inclui um editor, um compilador e um depurador.

Implementar uma interface Implementar uma classe que define todos os métodos especificados na interface.

Importando uma classe ou pacote Indicar a intenção de referenciar uma classe, ou todas as classes de um pacote, pelo nome simples em vez do nome qualificado.

Inicialização Configurar uma variável com um valor bem-definido quando ela é criada.

Inspeção passo a passo Executar um programa no depurador uma instrução por vez.

Instância de uma classe Objeto cujo tipo é essa classe.

Instanciação de uma classe Construção de um objeto dessa classe.

Instrução Unidade sintática em um programa. Em Java, uma instrução pode ser uma instrução simples, uma instrução composta ou um bloco.

Instrução simples Instrução que consiste somente em uma expressão.

Inteiro Número que não pode ter parte fracionária.

Interface Tipo sem variáveis de instância, composto apenas por constantes e métodos abstratos.

Interface pública Recursos (métodos, campos e tipos aninhados) de uma classe que são acessíveis a todos os clientes.

Internet Coleção mundial de redes, equipamentos de roteamentos e computadores que utilizam um conjunto comum de protocolos que definem a maneira como os participantes interagem entre si.

Interpretador Programa que lê um conjunto de códigos e executa os comandos especificados por eles.

Invariante de laço Instrução sobre o estado do programa que é preservada quando as instruções no laço são executadas uma vez.

Iterator Objeto que pode inspecionar todos os elementos em um contêiner, como uma lista encadeada.

Janela de inspeção Janela em um depurador que mostra os valores atuais de variáveis selecionadas.

Janela de shell Janela para interagir com um sistema operacional por meio de comandos textuais.

`javadoc` Gerador de documentação do SDK Java. Ele extrai comentários de documentação de arquivos-fonte Java e produz um conjunto de arquivos HTML vinculados.

JDK Kit de desenvolvimento de software Java que contém o compilador Java e ferramentas de desenvolvimento relacionadas.

Junção Consulta a banco de dados que envolve múltiplas tabelas.

JVM Java Virtual Machine (ou máquina virtual Java)

Laço Seqüência de instruções que é executada repetidamente.

Laço aninhado Laço contido em outro laço.

Lançar uma exceção Indicar uma condição anormal, terminando o fluxo de controle normal de um programa e transferindo o controle para uma cláusula `catch` correspondente.

Lei de De Morgan Lei sobre operações lógicas. Descreve como negar expressões formadas com operações *e* e *ou*.

Leiaute de borda Esquema de gerenciamento de leiaute em que componentes são colocados no centro ou em uma das quatro bordas de seu contêiner.

Leiaute de fluxo Esquema de gerenciamento de leiaute em que os componentes são dispostos da esquerda para a direita.

Leiaute de grade Esquema de gerenciamento de leiaute em que componentes são colocados em uma grade bidimensional.

Leitor Na biblioteca Java de entrada/saída, classe a partir da qual ler caracteres.

Limites assimétricos Limites que incluem o índice inicial mas não o índice final.

Limites simétricos Limites que incluem o índice inicial e o índice final.

Linguagem de criação de scripts Linguagem de programação que favorece o desenvolvimento rápido em detrimento da velocidade de execução e da capacidade de manutenção do código.

Linha de comando Linha onde o usuário digita para iniciar um programa em DOS, UNIX ou uma janela de comandos no Windows. Consiste no nome do programa seguido por qualquer argumento necessário.

Lista abstrata Seqüência ordenada de itens que pode ser percorrida seqüencialmente e que permite a inserção e remoção eficiente de elementos em qualquer posição.

Lista de arrays Classe Java que implementa um array de objetos com crescimento dinâmico.

Lista duplamente encadeada Lista encadeada em que cada nó possui uma referência tanto ao seu predecessor como ao seu sucessor.

Lista encadeada Estrutura de dados que pode armazenar um número arbitrário de objetos, cada um dos quais é armazenado em um objeto do tipo nó, que contém um ponteiro para o próximo link.

`main`, método Método que é inicialmente chamado quando uma aplicação Java é executada.

Map Estrutura de dados que mantém as associações entre chaves e valores.

Máquina de Turing Modelo muito simples de computação que é utilizado em ciência da computação teórica para explorar a computabilidade dos problemas.

Máquina virtual Programa que simula uma CPU que pode ser implementada eficientemente em uma variedade de máquinas reais. Um dado programa em bytecode Java pode ser executado por qualquer máquina virtual Java, independentemente de qual CPU é utilizada para executar a própria máquina virtual.

Marcação Informações sobre dados que são adicionados como instruções legíveis por humanos. Um exemplo são as marcas de documentos HTML com elementos como `<h1>` ou `<b>`.

Membro Método, campo ou tipo definido dentro de uma classe.

Mensagem de rastreamento Mensagem que é impressa durante a execução de um programa com propósitos de depuração.

Método Seqüência de instruções que tem um nome, pode ter parâmetros formais e retornar um valor. Um método pode ser invocado quantas vezes for necessário, com diferentes valores para seus parâmetros.

Método abstrato Método com um nome, tipos de parâmetros e tipo de retorno, mas sem uma implementação.

Método de acesso Método que acessa um objeto, mas não o altera.

Método de classe *Ver* **Método estático**

Método de instância Método com um parâmetro implícito; isto é, um método que é invocado em uma instância de uma classe.

Método de tratamento de evento Método que é executado quando um evento ocorre.

Método estático Método sem parâmetro implícito.

Método genérico Método com um ou mais parâmetros de tipo.

Método modificador Método que altera o estado de um objeto.

Método predicado Método que retorna um valor booleano.

Método recursivo Método que pode chamar a si próprio com valores mais simples. Ele tem de tratar os valores mais simples sem chamar a si próprio.

Modelo em cascata Modelo seqüencial de processo de desenvolvimento de software, que consiste em análise, projeto, implementação, testes e implantação.

Modelo em espiral Modelo iterativo de processo de desenvolvimento de software em que o projeto e a implementação são repetidos.

Módulo de tratamento de exceção Seqüência de instruções que é chamada quando uma exceção de um determinado tipo foi lançada e capturada.

`new`, operador Operador que aloca novos objetos.

Nome qualificado Nome que se torna não-ambíguo porque inicia com o nome de pacote.

Notação de ponto Notação *objeto*`.`*método*`(`*parâmetros*`)` ou *objeto*`.`*campo* utilizada para invocar um método ou acessar um campo.

Notação O grande (Big-Oh) Notação $g(n) = O(f(n))$, que indica que a função g cresce a uma taxa que é limitada pela taxa de crescimento da função f em relação a n. Por exemplo, $10n^2 + 100n - 1000 = O(n^2)$.

Notação polonesa invertida Estilo de escrever expressões em que os operadores são escritos depois dos operandos, como `2 3 4 * +` para indicar `2 + 3 * 4`.

Nova linha O caractere `'\n'`, que indica o fim de uma linha.

Número de ponto flutuante Número que pode ter uma parte fracionária.

Número mágico Número que aparece em um programa sem explicação.

Número pseudoaleatório Um número que parece ser aleatório, mas é gerado por uma fórmula matemática.

Object Valor de um tipo de classe.

Objeto anônimo Objeto que não é armazenado em uma variável especificada.

Operador Símbolo que indica uma operação matemática ou lógica, como `+` ou `&&`.

Operador binário Operador que aceita dois argumentos, por exemplo + em $x + y$.

Operador booleano *Ver* **Operador lógico**

Operador lógico Operador que pode ser aplicado a valores booleanos. Java tem três operadores lógicos: `&&`, `||` e `!`.

Operador pós-fixado Operador unário que é escrito depois de seu argumento.

Operador prefixado Operador unário que é escrito antes de seu argumento.

Operador relacional Operador que compara dois valores, produzindo um resultado booleano.

Operador ternário Operador com três argumentos. Java tem um operador ternário, `a ? b : c`.

Operador unário Operador com um argumento.

Oráculo Programa que prevê como outro programa deve se comportar.

Ordenação de dicionário *Ver* **Ordenação lexicográfica**

Ordenação lexicográfica Ordenar strings na mesma ordem que em um dicionário (alfabética), pulando todos os caracteres correspondentes e comparando os primeiros caracteres não-correspondentes de ambas strings. Por exemplo, "órbita" vem antes de "orquídea" na ordenação lexicográfica. Observe que, em Java, diferentemente de um dicionário, a ordenação faz distinção entre maiúsculas e minúsculas: `Z` vem antes de `a`.

Ordenação total Relacionamento de ordenação em que todos os elementos podem ser comparados entre si.

Ouvinte de evento Objeto notificado por uma fonte de evento quando um evento ocorre.

Pacote Coleção de classes relacionadas. A instrução `import` é utilizada para acessar uma ou mais classes em um pacote.

Painel Componente de interface com o usuário sem aparência visual. Ele pode ser utilizado para agrupar outros componentes.

Painel de conteúdo Parte de um frame Swing que armazena os componentes de interface com o usuário do frame.

Palavra reservada Palavra que tem um significado especial em uma linguagem de programação e, portanto, não pode ser utilizada como um nome pelo programador.

Parâmetro Item de informação que é especificado para um método quando o método é chamado. Por exemplo, na chamada `System.out.println("Hello, World!")`, os parâmetros são o parâmetro implícito `System.out` e o parâmetro explícito `"Hello, World!"`.

Parâmetro de tipo Parâmetro em uma classe genérica ou método que pode ser substituído por um tipo real.

Parâmetro explícito Parâmetro de um método que não corresponde ao objeto a partir do qual o método é invocado.

Parâmetro formal Variável em uma definição de método; ela é inicializada com um valor de parâmetro real quando o método é chamado.

Parâmetro implícito Objeto a partir do qual um método é invocado. Por exemplo, na chamada `x.f(y)`, o objeto `x` é o parâmetro implícito do método `f`.

Parâmetro real Expressão fornecida para um parâmetro formal de um método pelo chamador.

Passagem de parâmetro Especificar expressões como valores de parâmetro reais para um método quando ele é chamado.

Pasta *Ver* **Diretório**

Permutação Rearranjo de um conjunto de valores.

Pesquisa binária Algoritmo rápido para localizar um valor em um array classificado. Restringe a pesquisa à metade do array em cada passo.

Pesquisa linear Pesquisar um contêiner (como um array ou lista) em busca de um objeto, inspecionando um elemento de cada vez.

Pilha Estrutura de dados onde os dados são recuperados com base na regra "último a entrar, primeiro a sair". Os elementos podem ser adicionados e removidos somente em uma posição, chamada de topo da pilha.

Pilha de chamadas Conjunto ordenado de todos os métodos que foram chamados atualmente, mas não terminados ainda, iniciando com o método atual e terminando com `main`.

Pilha de tempo de execução Estrutura de dados que armazena as variáveis locais de todos os métodos chamados durante a execução de um programa.

Polimorfismo Selecionar um método entre vários métodos que têm o mesmo nome com base nos tipos reais dos parâmetros implícitos.

Ponteiro de arquivo Dentro de um arquivo de acesso aleatório, a posição do próximo byte a ser lido ou gravado. Ele pode ser movido a fim de acessar qualquer byte no arquivo.

Ponto de interrupção Ponto em um programa, especificado em um depurador, no qual este pára de executar o programa e deixa o usuário inspecionar o estado do programa.

Pós-condição Condição que é verdadeira depois que um método foi chamado.

Posição na memória Valor que especifica a localização de dados na memória do computador.

Precedência de operadores Regra que determina qual operador é avaliado primeiro. Por exemplo, em Java o operador `&&` tem uma precedência mais alta do que o operador `||`. Daí `a || b && c` ser interpretado como `a || (b && c)`.

Pré-condição Condição que deve ser verdadeira quando um método é chamado, se o método deve executar corretamente.

Programa de console Programa Java sem saída gráfica. Um programa de console lê entrada do teclado e exibe a saída na tela do terminal.

Programação Extrema Metodologia de desenvolvimento que busca simplicidade, removendo a estrutura formal e concentrando-se nas melhores práticas.

Programação genérica Fornecer componentes de programa que podem ser reutilizados em uma ampla variedade de situações.

Programação visual Programar organizando elementos gráficos em um formulário, configurar comportamentos de programas selecionando propriedades para esses elementos e gravar somente uma pequena quantidade de "código de cola" que os vincula.

Projeto Coleção de arquivos-fonte e suas dependências.

Projeto orientado a objetos Projetar um programa descobrindo objetos, suas propriedades e seus relacionamentos.

Prompt String que pede para o usuário fornecer uma entrada.

Quicksort Algoritmo de classificação geralmente rápido que seleciona um elemento, chamado pivô, particiona a seqüência em elementos menores e maiores do que o pivô e então classifica recursivamente as subseqüências.

RAM (Random-Access Memory) Memória de acesso aleatório. Circuitos eletrônicos em um computador que podem armazenar código e dados de programas em execução.

Rastreamento de pilha Impressão da pilha de chamadas, listando todas as chamadas de método atualmente pendentes.

Recursão Método para calcular um resultado decompondo as entradas em valores mais simples e aplicando o mesmo método a eles.

Recursão mútua Métodos colaboradores que chamam uns aos outros.

Recurso privado Recurso acessível somente por métodos da mesma classe ou uma classe interna.

Recurso protegido Recurso acessível a partir de uma classe, suas classes internas, suas subclasses e as outras classes no mesmo pacote.

Recurso público Recurso acessível por todas as classes.

Redirecionamento Associar a entrada ou saída de um programa a um arquivo em vez do teclado ou o monitor.

Referência a objeto Valor que indica a localização de um objeto em memória. Em Java, uma variável cujo tipo é uma classe contém uma referência a um objeto dessa classe.

Referência nula Referência que não referencia nenhum objeto.

Registro em log Envio de mensagens que rastreiam o progresso de um programa para um arquivo ou janela.

Script de shell Arquivo que contém comandos para executar programas e manipular arquivos. Digitar o nome do arquivo de script de shell na linha de comando faz com que esses comandos sejam executados.

Sentinela Valor na entrada que não é utilizado como um valor de entrada real, mas para sinalizar o fim da entrada.

Serialização Processo de salvar um objeto, e todos os objetos que ele referencia, em um fluxo.

Servidor Programa ou sistema de computador que recebe solicitações de um cliente, obtém ou calcula as informações solicitadas e as envia para o cliente.

Set Coleção não-ordenada que permite adição, localização e remoção de elementos de modo eficiente.

Shell Parte de um sistema operacional onde o usuário digita as linhas de comando para executar programas e manipular arquivos.

Sintaxe Regras que definem como compor instruções em uma determinada linguagem de programação.

Sistema operacional Software que carrega programas aplicativos e fornece serviços (como um sistema de arquivos) para esses programas.

Sobrecarregar Dar mais de um significado para um nome de método.

Sobrescrever Redefinir um método em uma subclasse.

Sombreamento Ocultar uma variável definindo outra com o mesmo nome.

String Seqüência de caracteres.

Stub Método com funcionalidade mínima ou nenhuma funcionalidade.

Subclasse Classe que herda variáveis e métodos de uma superclasse, mas adiciona variáveis de instância e métodos ou redefine métodos.

Superclasse Classe geral da qual uma classe mais especializada (uma subclasse) herda elementos.

Swing Conjunto de ferramentas Java para implementar interfaces gráficas com o usuário.

Tabela hash Estrutura de dados onde elementos são mapeados para posições de array de acordo com seus valores de função de hash.

TAD (Tipo Abstrato de Dados) Uma especificação das operações fundamentais que caracterizam um tipo de dados, sem fornecer uma implementação.

TCP/IP (Transmission Control Protocol/Internet Protocol) Protocolo de controle de transmissão/Protocolo de Internet. Par de protocolos de comunicação que é utilizado para estabelecer transmissão confiável de dados entre dois computadores na Internet.

Teste da caixa preta Testar funções considerando suas implementações, ao contrário do teste da caixa preta; por exemplo, selecionando casos de teste de limite e assegurando que todos os desvios do código sejam cobertos por algum caso de teste.

Teste da caixa preta Testar um método sem conhecer sua implementação.

Teste de regressão Manter casos de teste antigos e testar cada revisão de um programa com eles.

Teste de unidade Teste de um método sozinho, isolado do restante do programa.

`throws`, especificador Indica os tipos das exceções verificadas que um método pode lançar.

Tipo Conjunto definido de valores e as operações que podem ser executadas sobre eles.

Tipo booleano Tipo com dois valores possíveis: `true` e `false`.

Tipo enumerado Tipo de dados com um número finito de valores, cada um dos quais com seu próprio nome simbólico.

Tipo primitivo Em Java, um tipo numérico ou `boolean`.

Token Seqüência de caracteres consecutivos de uma fonte de entrada, separados com o propósito de analisar a entrada. Por exemplo, um token pode ser uma seqüência de caracteres diferentes do espaço em branco.

`try`, bloco Bloco de instruções que contém cláusulas de processamento de exceção. Um bloco `try` contém pelo menos uma cláusula `catch` ou `finally`.

Um laço e meio Laço cuja decisão de término não está no início nem no fim.

Unicode Código padrão que atribui valores de código de dois bytes para representar caracteres utilizados em scripts em todo o mundo. Java armazena todos os caracteres como valores Unicode.

Unified Modeling Language (UML) Notação para especificar, visualizar, construir e documentar os artefatos de sistemas de software.

URL (Uniform Resource Locator) Ponteiro para um recurso de informação (como uma página Web ou uma imagem) na Web.

Valor de retorno Valor retornado por um método através de uma instrução `return`.

Variável Símbolo em um programa que identifica uma posição na memória que pode armazenar diferentes valores.

Variável de parâmetro Variável de um método que é inicializada com um valor de parâmetro quando o método é chamado.

Variável de tipo Variável na declaração de um tipo genérico que pode ser instanciada com um tipo.

Variável local Variável cujo escopo é um bloco.

Variável não-inicializada Variável que não foi configurada com um valor particular. Em Java, utilizar uma variável local não-inicializada gera um erro de sintaxe.

Vinculação inicial Escolha em tempo de compilação entre os vários métodos com o mesmo nome, mas tipos de parâmetro diferentes.

Vinculação tardia Escolher em tempo de execução entre vários métodos com o mesmo nome invocados a partir de objetos que pertencem a subclasses da mesma superclasse.

`void`, palavra-chave Palavra-chave que indica nenhum tipo ou um tipo desconhecido.

Índice

As referências de página seguidas por *t* indicam material em tabelas. As, classes da biblioteca Java são indexadas sob `java`, por exemplo "`java.util.Scanner`,, classe."

! (operador lógico *não*) 206, 207
  e lei de De Morgan 209
&& (operador lógico *e*) 206, 207
  avaliação preguiçosa 209
  confundindo com || 207–208
  e lei de De Morgan 209
|| (operador lógico *ou*) 206, 207
  avaliação preguiçosa 209
  confundindo com && 207–208
  e lei de De Morgan 209
!=, não use para testar o fim de intervalo em laços 237
%, operador 157
+, operador 166
=, operador, *Ver* atribuição (=), operador de
!= , operador 191*t*
/, operador 156–157
<, operador 191*t*
<=, operador 191*t*
==, operador 191*t*
  comparando objetos 193–195
  não use para comparar strings 192–194
  testando a referência null 194–196
>, operador 191*t*
>=, operador 191*t*

A

abs, método
  classe `java.lang.Math` 158*t*, 662
abstração 100–103
  níveis em projeto de software 101–102
Abstract Windowing Toolkit (AWT) 74–75, 339*t*
`AccountTester.java` 418
acesso aleatório
  listas encadeadas 604, 621, 622
acesso de pacote 423–424
  acidental 424
acesso protegido 422, 425
acesso seqüencial
  listas encadeadas 604, 621, 622
acoplamento 319–320
  interfaces para reduzir 363
acos, método
  classe `java.lang.Math` 158*t*, 662
`actionPerformed`, método
  interface `java.awt.event.ActionListener` 379–380, 382, 384–385, 653
adaptadores de evento 388
add, método
  classe `java.awt.Container` 650
  classe `java.math.BigDecimal` 667
  classe `java.math.BigInteger` 667
  classe `java.util.ArrayList` 276–278, 667
  classe `javax.swing.ButtonGroup` 674–675
  classe `javax.swing.JMenu` 676–677
  classe `javax.swing.JMenuBar` 676–677
  interface `java.util.Collection` 668
  interface `java.util.ListIterator` 607, 671
`addActionListener`, método
  classe `javax.swing.AbstractButton` 673–674
`addChangeListener`, método
  classe `javax.swing.JSlider` 677–678
`addFirst`, método
  classe `java.util.LinkedList` 605, 610–611, 670
`addItem`, método
  classe `javax.swing.JComboBox` 674–675
`addLast`, método
  classe `java.util.LinkedList` 605, 670
`addMouseListener`, método
  classe `java.awt.Component` 650

`Address.java` 502–503
adição 156
agregação 488
  multiplicidades 491
  notação UML 489
  símbolo de relacionamento UML 490*t*
agulha de Buffon, experiência da 245–247
ajuda on-line 77–78
alfabetos internacionais 169
algoritmos de classificação 572
  por inserção 581
  por intercalação 581–584
  por seleção 572–581
  quicksort 588
algoritmos para arrays 285–289
alto-falantes 34–35
ambiente de desenvolvimento integrado 40
animação gráfica 601
anulando métodos 407
API, documentação da 75–78
aplicativos gráficos 79–80
  colocando texto 86–87
  cores 86–88
  desenhando formas 81–85
  desenhando, formas complexas 87–89, 126–134
  janelas de frame 79–82
`append`, método
  classe `javax.swing.JTextArea` 437, 677–678
applet para visualização de moléculas 38–39
applets 38–39, 84–85
arcos, desenhando 82–83
argumentos de linha de comando 46, 455
armazenamento de dados 32–35
armazenamento primário 32–33
armazenamento removível 34–35
armazenamento secundário 32–33
arquivos 40–42
  lendo e gravando arquivos de texto 452–455
arquivos de classe 51–52
arquivos em lote 302
arquivos-fonte, guia de estilo 636
`arraycopy`, método
  classe `java.lang.System` 294–296, 666
`ArrayList`, construtor 667
`ArrayListTester.java` 278–279
arrays 272–276
  algoritmos simples 285–289
  bidimensionais 290–294
  bidimensionais com comprimentos de linha variáveis 293
  circular 625, 630
  Como Fazer 293
  copiando 295–300
  eficiência de operações 622*t*
  empacotadores e auto-empacotamento 281–283
  inicialização 276
  multidimensionais 294
  não-inicializados 276
  número de elementos 281*t*
  parcialmente preenchidos 299
  subestimando o tamanho do conjunto de dados 298
  transformando arrays paralelos em arrays de objetos 298–299
  Ver também classificando
`ArrayUtil.java` 575
arredondamento, erros de 147, 163
árvores de sintaxe 555–556
`asin`, método
  classe `java.lang.Math` 158*t*, 662
assertiva 327–328
associação 491
`atan`, método
  classe `java.lang.Math` 158*t*, 662
`atan2`, método
  classe `java.lang.Math` 158*t*, 662
`ATM.java` 512–514
`ATMFrame.java` 519–521
`ATMSimulator.java` 517–518
`ATMViewer.java` 518–519
atores 316–317
atribuição (=), operador de 62–63, 155–156
  combinando com aritmética 156
atributo protegido 637
atributos em diagramas UML 491
auto-empacotamento 282–283
avaliação preguiçosa 209

B

Babbage, Charles 588
`BadDataException.java` 471
`Bank.java` 287–289, 514–515
`BankAccount.java` 114–116, 279–280, 419–420
`BankAccountTester.java` 119–121
`BankTester.java` 289, 300–301
barramento 34–35

barras de rolagem 437
barras invertidas em nomes de arquivo 454
Basic Latin (ASCII) subconjunto, do Unicode 644*t*
biblioteca Java 37–38, 39
  ajuda on-line 77–78
  documentação da API 75–78
  herança, hierarquia 646–648
  listagem completa por pacote 648–679
`BigDecimal`, construtor 666
`BigInteger`, construtor 667
`binarySearch`, método
  classe `java.util.Arrays` 593, 668
  classe `java.util.Collections` 669
`BinarySearcher.java` 591–592
`BLACK`, valor RGB 87–88*t*
bloco 381
bloco de instruções 188, 189
blocos de inicialização 334
`BLUE`, valor RGB 87–88*t*
BlueJ 74–75, 118–119
  depurador integrado 248
`Boolean`, classe 282*t*
`Boolean`, construtor 660
`boolean`, tipo 147*t*
  classe empacotadora 282*t*
  utilizando 205
`booleanValue`, método
  classe `java.lang.Boolean` 283, 660
`BorderLayout`, construtor 649
botões, construindo aplicações com 384–388
`break`, comando 245
  quando desativado pelo guia de estilo 636, 640
bugs 49–50
  primeiro bug 258
  Ver também depuradores
`ButtonViewer.java` 380
`Byte`, classe 282t
`byte`, tipo 147*t*
  classe empacotadora 282*t*

C

C
  erros de limite 275
C++
  erros de limite 275
  Java, comparado com 38–39
  palavra-chave `static` 330
  rival mais próximo de Java 37–38
caixas de diálogo, lendo entrada a partir de 172
caixas de diálogo de arquivo 454
caixas pretas 100–101
calculando/computando 35
caminho de classe 341, 342
campos 110–111
campos de classe 332
campos de instância 110–113, 121–124, 637
  classes e métodos abstratos 422
  clone modificável em métodos de acesso 431
  determinando 117–118
  parâmetro implícito 124–125
  sombreando 411–412
  subclasses 401
campos de texto 433–434
  interfaces gráficas com usuário 437–439
campos estáticos 331–334, 637
capturando exceções 460–463
`Car.java` 129–130
caracteres de controle 643*t*
caracteres de escape 454
caracteres suplementares 169
`CarComponent.java` 128–129
`CarViewer.java` 130
`CashRegister.java` 152–153
`CashRegisterSimulator.java` 170–171
`CashRegisterTester.java` 153
casos de teste 203–204
  calculando dados de exemplo manualmente 211–212
  limite 211
  preparando antecipadamente 212
`catch`, cláusula 461–463
  não utilizar `catch` e `finally` no mesmo comando `try` 465
`catch`, instrução 460–463
  utilizando espaços em torno de (. . .) parte 641
CDs (compact discs) 34–35
  salvando backups em 42
`ceil`, método
  classe `java.lang.Math` 158*t*, 662
chamada 47, 48
chamada de método 47, 48
chamada por referência 326
chamada por valor 326
`char`, tipo 147*t*
  classe empacotadora 282*t*
  e strings 169

`Character`, classe 282*t*
chaves 46
  alinhando 189–190
  formando blocos de instruções 188
  guia de estilo 636, 642
`CheckingAccount.java` 420–421
chips 32–33
chips de memória 32–34
ciclo 300
ciclo de vida de software 480–484
circuitos integrados 32–33
círculos
  Como Fazer 131
  desenho 85–86
classe `java.lang.Double` 283, 661
classe testadora 119–120
classe utilitária 316–317
classes 44, 59, 63–65
  anônimas 377
  campos de instância 110–113, 117–118
  coesão e acoplamento 318–321
  comentando 107–109
  Como Fazer 115–119
  construindo para realizar cálculos 163–165
  controle de acesso 422–425
  descobrindo 485–487
  escolhendo 316–317
  escopo de membros 335–336
  guia de estilo 637
  implementando construtores e métodos 112–119
  imutáveis 321
  localização 341–342
  método da ficha CRC 486–487
  projetando interface pública 102–108, 116–117
  relacionamentos entre 485, 488–491
  testando 118–119
  Ver também herança; classes internas
classes abstratas 422
classes candidatas 485–486
classes de exceção, hierarquia de 456
classes de tratamento de eventos 439–440
classes genéricas 276–277
classes internas 85–86, 375–377, 637
  como ouvintes de evento 381–383
  e reutilização 359
  e variáveis adjacentes 381–382
  estáticas 619
classificação por inserção 581
classificação por intercalação 581–584
  analisando 584–587
classificação por seleção 572–575
  análise de desempenho 579–581
  perfil 575–579
classificando 571
  dados reais 593–596
`ClickListener.java` 379
cliente no local 484
`clone`, método
  classe `java.lang.Enum` 431
  classe `java.lang.Object` 425*t*, 430–431, 664
  cópia de array 294–296
  esquecendo do 430–431
  implementando 431
`close`, método
  classe `java.io.InputStream` 656
  classe `java.io.OutputStream` 657
  classe `java.io.PrintWriter` 452, 463, 658
  classe `java.io.Reader` 659
  classe `java.io.Writer` 659
  classe `java.util.Scanner` 452, 672
cobertura do teste 210–213
código de máquina
  código espaguete 230
  fatorando código comum 161
  traduzindo programas de computador em 36–38
código-fonte 51–52
códigos de hash
  e sobrescrevendo método `toString` 426
coerção 148
  implementando listas encadeadas 609
  para conversão do tipo de interface em tipo de classe 368
  para recuperação do objeto `Graphics2D` a partir do parâmetro `Graphics` 82–83
coesão 318–321
colaboradores, em ficha CRC 486–487
colchetes 641
coleções
  guia de estilo 639
  utilizando parâmetros de tipo, não tipos brutos 639
coletor de lixo 121–122
  listas encadeadas 610
`Color`, construtor 649
comando de um único passo 250
comandos 46

começo da fila 623
comentários 46
  guia de estilo 636
  sintaxe alternativa 48–49
Common Object Request Broker Architecture (CORBA) 339*t*
`Comparable`, interface parametrizada 595–596
comparação lexicográfica 192–194
comparações
  comparando valores 191–196
  seqüências de 196–198
`Comparator`, interface 596
`compare`, método
  interface `java.util.Comparator` 669
`compareTo`, método
  classe `java.lang.String` 192–194, 665
  interface `Comparable` 594, 595, 660
  retornando qualquer inteiro 595
compilador
  localizando 40
  traduzindo programas em código de máquina 36–38
compilando
  laço editar-compilar-testar 52–53
  processo de compilação 50–53
  programa simples 43–48
componentes
  adicionando a frames, ao desenhar formas 81–85
  largura e altura zero por padrão 441
computação recursiva 535
computação/cálculo
  Como Fazer 163–165
  limites de 554
computador
  anatomia de 31–35
  conhecendo 40–43
  diagrama esquemático 35
  mainframes 79–80
  necessidade de programação 30
  pessoal 345
concatenação 166, 426
condições
  comando `if` 186
  evitando efeitos colaterais 195–196
  laços `do` 230
  laços `while` 226
condições de erro 203–204
conjunto de testes 300
consistência 320
constantes 149–154
  em interfaces 366
  guia de estilo 638–639
  guia de estilo para nomear 636, 640
constantes estáticas 333, 338
constantes numéricas 149
construção 70–71
  arrays 274
construtores 69–72, 104–107, 637
  chamando um a partir do outro 126–127
  esquecendo de inicializar referências a objeto 123–124
  implementando 112–119
  métodos, comparado com 71–72
`contains`, método
  interface `java.util.Collection` 668
contêineres, não utilizar como um ouvinte de evento 387–388
`continue`, comando 245
  quando desativado pelo guia de estilo 636, 640
controladora de disco 35
controle de acesso 422–425
  acesso protegido 425
copiar e colar, em editor 198–199
cópias de backup 42
cores 86–88
  valores RGB predefinidos 87–88*t*
corpo
  comando `if` 186
  do método 46, 103–104
`cos`, método
  classe `java.lang.Math` 158*t*, 662
`currentTimeMillis`, método
  classe `java.lang.System` 576, 666
`Customer.java` 516
`CYAN`, valor RGB 87–88*t*

D

`DARKGRAY`, valor RGB 87–88*t*
`DataAnalyzer.java` 240, 467–468
`DataSet.java` 240–241, 372–373
`DataSetReader.java` 470–471
`DataSetTester.java` 365, 373
`DataSetTester3.java` 376–377
De Morgan, lei de 209
decisões 186
  comparando valores 191–196
  desvios aninhados 198–202

várias alternativas 196–205
Ver também `if`, comando
decremento (`--`), operador de 155–156
dependência 318–320, 489
símbolo de relacionamento UML 490*t*
depuradores 49–50
Como fazer, sessão de exemplo 257–258
métodos recursivos 542
não podem mostrar que o programa está livre de bugs 256
preferidos a mensagens registradas em log 213
utilizando 248–251
desenvolvimento de software 480–484
arte ou ciência? 524
modelo em cascata 481–482
modelo em espiral 482–483
`destroy`, método
classe `java.applet.Applet` 649
desvios aninhados
comando `if` 198–202
diagrama de sintaxe de expressão 555
diagrama de sintaxe de fator 555
diagrama de sintaxe de termo 555
diagramas de classe 319
diagramas de objeto 319
diagramas de sintaxe 555–556
`Dimension`, construtor 650
diretório atual 342
diretório de base 341–342, 344, 345
diretórios 40–41
diretrizes de codificação 635–642
discos rígidos 32–35
dispositivos de entrada 34–35
dispositivos de saída 34–35
disquetes 34–35
salvando backups em 42
distinção entre maiúsculas e minúsculas
de identificadores 61
de Java 43
`divide`, método
classe `java.math.BigInteger` 667
dividir para conquistar, técnica de depuração 257
divisão 156
divisão de inteiro 159–160
`do`, laços 230
`break` e `continue` 245
documentação de método
estudo de caso da impressão de fatura 496–498
estudo de caso do caixa automático 510
documentação, comentários 107–108
documento de requisitos 480
estudo de caso da impressão de fatura 492
estudo de caso do caixa automático 503–505
`Double`, classe 282*t*
`Double`, construtor 661
`double`, tipo 68–69, 147*t*
classe empacotadora 282*t*
não apropriado para cálculos financeiros 147–148
`doubleValue`, método
`draw`, método
classe `java.awt.Graphics2D` 82–83, 85–87, 133, 651
`drawString`, método
classe `java.awt.Graphics2D` 86–87, 651

E

`E`, constante, classe `java.lang.Math` 151
e, operador lógico 206, 207
avaliação preguiçosa 209
confundindo com o operador *ou* 207–208
e lei de De Morgan 209
`Earthquake.java` 196–198
`EarthquakeRunner.java` 197–199
Eclipse 248
JUnit incorporado 345
edição corporativa da biblioteca Java 37–38
edição de linha de comando 120–121
edição micro, da biblioteca Java 37–38
editar-compilar-testar, laço 52–53
editor 51–52
copiar e colar em 198–199
efeitos colaterais
de condições 195–196
de métodos 321–322
minimizando 322, 325
efetuando login 40
elipses
Como Fazer 131, 132
desenhando 82–83
`Ellipse2D.Double`, construtor 653
`else` oscilante, problema do 203–204
`else`, alternativa 187, 188
problema do `else` oscilante 203–203–204
empacotadores 281–283
`EmptyFrameViewer.java` 80–82
encapsulamento 100–101, 112–113
ENIAC (Electronic Numerical Integrator and Computer) 35

entrada
lendo 169–172
lendo a partir de caixa de diálogo 172
processando entrada de texto 433–436
`EOFException`, construtor 655
epsilon, limiar para números de ponto flutuante 191–192
`equals`, método
anulando 427–429
classe `java.lang.Enum` 431
classe `java.lang.Object` 193–195, 425*t*, 664
classe `java.lang.String` 192–193, 665
definindo com tipo errado de parâmetro 429
e herança 429
`equalsIgnoreCase`, método
classe `java.lang.String` 192–193, 665
erro da estaca de cerca 245
Erro de assertiva 327
erro de ortografia 50–51
erro de sintaxe 48–50, 52–53
erro de variável não-inicializada 63
erros 48–51
reproduzindo em depuradores 257
verificando no processo de compilação 52–53
Ver também depuradores
erros de limite 274, 275
erros de lógica 49–50
erros em tempo de compilação 48–50
escopo
membros de classe 335–336
sobrepondo 336–337
variáveis locais 334–335
espaço em branco 160–161, 170
guia de estilo 641
espaços
guia de estilo para utilizar 636, 641
Unicode para 643*t*
espaguete, código 230
especificação de requisito 481
especificador de acesso
campos de instância 111–112
método 103–104
estratégia de backup 42, 43
estruturas de dados 603, 604
pilhas e filas 623–625
tipos abstratos e concretos de dados 619–622
Ver também pesquisa binária; listas encadeadas
estudo de caso do caixa eletrônico 503–524
`EtchedBorder`, construtor 678–679
`Evaluator.java` 558–559
eventos 377–378
de processamento de timers 388
eventos de mouse 388
exceções
guia de estilo 640
lançando específicas 466–467
não silenciar 463
projetando seus próprios tipos 465–467
exceções não-verificadas 458–460
exceções verificadas 458–460
`exists`, método
classe `java.io.File` 655
`exit`, método
classe `java.lang.System` 666
`exp`, método
classe `java.lang.Math` 158*t*, 663
`ExpressionCalculator.java` 560–561
`ExpressionTokenizer.java` 559–560
expressões booleanas 205–209
expressões regulares 338
extensão 40

F

`FaceComponent.java` 87–89
`FaceViewer.java` 88–90
`false`, tipo `boolean` 206
fase de análise, do ciclo de vida de software 480
descoberta de classe em 486
fases do ciclo de vida do software 480–481
fatura, estudo de caso da impressão de 491–503
ferramenta JUnit 345–347
Fibonacci, seqüência de 263–264, 548
filas 623–625
limites de 630
`File`, construtor 655
`FileInputStream`, construtor 655
`FileNotFoundException` 453, 459, 468, 655
`FileOutputStream`, construtor 656
`FileReader`, construtor 656
`FileWriter`, construtor 656
`fill`, método
classe `java.awt.Graphics2D` 87–88, 651
final da fila 623
final, classes 422
final, métodos 422
`finally`, cláusula 463–465
não utilizar `catch` e `finally` na mesma instrução `try` 465

fitas para armazenamento de dados 34–35
flags
  variáveis booleanas 209
float tipo 147*t*
  classe empacotadora 282*t*
Float, classe 282*t*
floor, método
  classe java.lang.Math 158*t*, 663
FlowLayout, construtor 651
fluxo de controle
  guia de estilo 639–640
fluxo de controle não-linear
  guia de estilo 640
Font, construtor 651
fontes de evento 377–378
for each, laços 284–285, 639
  e interface Iterable 608
for, instrução
  guia de estilo 639–640
  utilizando espaços em torno de (. . .) parte 641
for, laços 231–237
  aprimorados 283–285
  break e continue 245
  Como Fazer 243
  contando número de iterações 244–245
  escopo de variáveis definido no cabeçalho 237
  uso somente para propósito concebido 235
formas 84–89
  Como Fazer 131–134
  desenhando 81–85
  desenhando complexas 87–89, 126–134
formas gráficas 84–89
  Como Fazer 131–134
  desenhando complexas 87–89, 126–134
format, método
  classe java.lang.String 665
frames 80–82
  desenhando formas em 81–85
  utilizando herança para personalizar 432–434
função de base 164

G

gerador de número aleatório 246
get, método
  classe java.util.ArrayList 276–277, 622, 667
  classe java.util.Calendar 668
  interface java.util.Map 671
getCenterX, método
  classe java.awt.geom.RectangularShape 655
getCenterY, método
  classe java.awt.geom.RectangularShape 655
getFilePointer, método
  classe java.io.RandomAccessFile 659
getFirst, método
  classe java.util.LinkedList 605, 670
getHeight, método
  classe java.awt.Component 128–129, 650
  classe java.awt.geom.RectangularShape 654
  classe java.awt.Rectangle 71–72, 652
getLast, método
  classe java.util.LinkedList 605, 670
getLogger, método
  classe java.util.logging.Logger 213, 673–674
getMaxX, método
  classe java.awt.geom.RectangularShape 655
getMaxY, método
  classe java.awt.geom.RectangularShape 655
getMessage, método
  classe java.lang.Throwable 466, 666
getMinX, método
  classe java.awt.geom.RectangularShape 655
getMinY, método
  java.awt.geom.RectangularShape classe 655
getSelectedFile, método
  classe javax.swing.JFileChooser 675–676
getSelectedItem, método
  classe javax.swing.JComboBox 674–675
getSource, método
  classe java.util.EventObject 669
getText, método
  classe javax.swing.JTextComponent, 433–434, 679
getValue, método
  classe javax.swing.JSlider 677–678
getWidth, método
  classe java.awt.Component 128–129, 650

classe `java.awt.geom.RectangularShape`
classe `java.awt.Rectangle` 71–72, 652
`getX`, método
classe `java.awt.event.MouseEvent` 653
classe `java.awt.geom.Point2D` 654
classe `java.awt.Rectangle` 71–72, 652
`getX1`, método
classe `java.awt.geom.Line2D` 654
`getX2`, método
classe `java.awt.geom.Line2D` 654
`getY`, método
classe `java.awt.event.MouseEvent` 653
classe `java.awt.geom.Point2D` 654
classe `java.awt.Rectangle` 71–72, 652
`getY1`, método
classe `java.awt.geom.Line2D` 654
`getY2`, método
classe `java.awt.geom.Line2D` 654
`goto` comando 230
`GRAY`, valor RGB 87–88*t*
`GREEN`, valor RGB 87–88*t*
`GregorianCalendar`, construtor 669
`GridLayout`, construtor 652
guia de estilo 635–642

H

`HashMap`, construtor 670
`HashSet`, construtor 670
`hasNext`, método
classe `java.util.Scanner` 672
interface `java.util.Iterator` 607, 670
`hasNextDouble`, método
classe `java.util.Scanner` 206, 672
`hasNextInt`, método
classe `java.util.Scanner` 206, 672
`hasNextLine`, método
classe `java.util.Scanner` 672
`hasPrevious`, método
interface `java.util.ListIterator` 607, 671
Hello, World! 43–48
`HelloPrinter.java` 43–48
herança 400–404, 488
campos de instância e métodos 407–412
e método `equals` 429
e método `toString` 427
notação UML 489
personalizando frames 432–434
símbolo de relacionamento UML 490*t*
herança, hierarquias 405–407
HTML, documentação 110–111

I

identificador 61
`if`, comando 186–191
comparando valores 191–196
desvios aninhados 198–202
guia de estilo 639
problema do `else` oscilante 203–204
utilizando espaços em torno de (. . .) parte 641
`if/else`, comando 187–188
`IllegalArgumentException` 455, 457, 661
`IllegalStateException` 455
`ImageIcon`, construtor 674–675
imagens gráficas computadorizadas 135
implementação 102–103
tipos de interface 362–363, 364
implementação privada 64–65
`import`, instrução 636
implementando um programa de teste 74–75
pacotes Java 340
importações estáticas 338
impressora 34–35
incidente do foguete Ariane 472
`info`, método
classe `java.util.logging.Logger` 213, 673–674
informando e lançando exceções 455
inicialização
arrays 276
campos estáticos 332
esquecendo de inicializar referências a objeto 123–124
formas alternativas de campo 334
`init`, método
classe `java.applet.Applet` 649
`InputStreamReader`, construtor 657
`instanceof`, operador 416
instanciação
interfaces 368
instrução composta 188
instrução de pacote 636
instrução simples 188
instruções de máquina 36–38
`int`, tipo 68–69, 147*t*
classe empacotadora 282*t*
`Integer`, classe 282*t*
`Integer`, construtor 661
integração contínua 484
inteiros 68–69

Intel Corporation
bug de ponto flutuante do Pentium 149
inteligência artificial 210
interface com o usuário
Como Fazer 439–440
eventos, fontes de evento e eventos ouvintes 377–381
pacote de design 39
protótipo 482–483
interface implementação de
estudo de caso da caixa automático 511–524
estudo de caso da impressão de fatura 498–503
símbolo de relacionamento UML 490*t*
interface `java.util.Collection`
resumo do método 668–669
interface `java.util.Comparator`
resumo do método 669
interface `java.util.Map`
resumo do método 671
interfaces 359
constantes em 366
guia de estilo para nomear 641
instanciação 368
para reutilização de código 360–365
processando eventos de timer 388
utilizando, para retornos de chamada 370–374
interfaces gráficas com usuário (GUIs)
áreas de texto 437–439
Como Fazer 439–440
utilizando herança para personalizar frames 432–434
Internet 35
evolução da 89–90
salvando backups na 42
`intersection`, método
classe `java.awt.Rectangle` 652
`intValue`, método
classe `java.lang.Integer` 283, 661
invariantes de classe 329
invariantes do laço 247
`Investment.java` 233–234
`InvestmentFrame.java` 434–439
`InvestmentRunner.java` 234
`InvestmentViewer1.java` 382–383
`InvestmentViewer2.java` 385–386
`InvestmentViewer3.java` 434–435
`Invoice.java` 500–501
`InvoicePrinter.java` 499
`isDigit`, método
classe `java.lang.Character` 206, 660
`isEditable`, método
classe `javax.swing.JComboBox` 674–675
classe `javax.swing.JTextComponent` 679
`isLetter`, método
classe `java.lang.Character` 206, 660
`isLowerCase`, método
classe `java.lang.Character` 206, 660
`isSelected`, método
classe `javax.swing.AbstractButton` 673–674
`isUpperCase`, método
classe `java.lang.Character` 206, 660
`Iterable`, interface 608
iteração 225
Ver também `for`, laços; `while`, laços; laços
iteradores de lista 605
`iterator`, método
interface `Collection` 669

J

janela de shell 40
Java
como linguagem de programação de alto nível 36–38
guia de estilo de codificação 635–642
história da versão 39*t*
história do desenvolvimento 37–39
não projetado especificamente para alunos 39
padronização 626
Java 1.0 39*t*
Java 1.1 39*t*
Java 1.2 39*t*
Java 1.3 39*t*
Java 1.4 39*t*
Java 5.0 39, 39*t*
listas de arrays 283
nova classe `Scanner` 170
Java 6 39*t*
Java, applets 38–39, 84–85
`.java`, extensão 40
Java, pacotes 39, 339–340
atribuição de nomes 341
importando 340
importando classes de 74–75
listagem de classes por pacote 648–679
Java, programas 42
`java.applet`, pacote 339*t*, 649

`java.applet.Applet`, classe
resumo do método 649
`java.awt`, pacote 339*t*, 649–652
`java.awt.BorderLayout`, classe
resumo do método 649
`java.awt.Color`, classe
para aplicativos gráficos 86–88
resumo do método 649
`java.awt.Component`, classe
resumo do método 650
`java.awt.Container`, classe
resumo do método 650
`java.awt.Dimension`, classe
resumo do método 650
`java.awt.event`, pacote 653
`java.awt.event.ActionEvent`, interface 382
`java.awt.event.ActionListener`, interface
botões 384–385
resumo do método 653
`java.awt.event.MouseEvent`, classe
resumo do método 653
`java.awt.event.MouseListener`, interface
resumo do método 653
`java.awt.FlowLayout`, classe
resumo do método 651
`java.awt.Font`, classe
resumo do método 651
`java.awt.Frame`, classe
resumo do método 651
`java.awt.geom`, pacote 653–655
`java.awt.geom.Ellipse2D.Double`, classe
85–86
resumo do método 653
`java.awt.geom.Ellipse2D.Float`, classe
85–86
`java.awt.geom.Line2D`, classe
resumo do método 654
`java.awt.geom.Line2D.Double`, classe
linhas de desenho 85–87
resumo do método 654
`java.awt.geom.Point2D`, classe
resumo do método 654
`java.awt.geom.Point2D.Double`, classe
resumo do método 654
`java.awt.geom.Rectangle2D.Double`, classe
85–86
`java.awt.geom.RectangularShape`, classe
resumo do método 654–655
`java.awt.Graphics`, classe 82–84
resumo do método 651
`java.awt.Graphics2D`, classe 82–84
resumo do método 651
`java.awt.GridLayout`, classe
resumo do método 652
`java.awt.Rectangle`, classe
construindo objetos 69–72
documentação da API para 75–77
implementando programa de teste 72–75
resumo do método 652
`java.awt.Shape`, interface
resumo do método 652
`java.io`, pacote 339*t*, 655–659
`java.io.EOFException`, classe 456
resumo do método 655
`java.io.File`, classe
resumo do método 655
`java.io.FileInputStream`, classe
resumo do método 655
`java.io.FileNotFoundException`, classe 456
capturando 461, 462
resumo do método 655
`java.io.FileOutputStream`, classe
resumo do método 656
`java.io.FileReader`, classe
lendo e gravando arquivos de texto 452
resumo do método 656
`java.io.FileWriter`, classe
resumo do método 656
`java.io.InputStream`, classe
resumo do método 656
`java.io.InputStreamReader`, classe
resumo do método 657
`java.io.IOException`, classe 456
como exceções verificadas 458
resumo do método 657
`java.io.ObjectInputStream`, classe
resumo do método 657
`java.io.ObjectOutputStream`, classe
resumo do método 657
`java.io.OutputStream`, classe
resumo do método 657
`java.io.PrintStream`, classe 51–52, 63–64
resumo do método 657–658
`java.io.PrintWriter`, classe
lendo e gravando arquivos de texto 452
resumo do método 658
`java.io.RandomAccessFile`, classe
resumo do método 659
`java.io.Reader`, classe
resumo do método 659

`java.io.Writer`, classe
  resumo do método 659
`java.lang`, pacote 660–666
`java.lang.ArithmeticException`, classe 456
`java.lang.ArrayIndexOutOfBounds-Exception`, classe 456
`java.lang.Boolean`, classe
  resumo do método 660
`java.lang.Character`, classe
  resumo do método 660
`java.lang.ClassCastException`, classe 456
`java.lang.ClassNotFoundException`, classe 456
`java.lang.Cloneable`, interface
  resumo do método 660
`java.lang.CloneNotSupportedException`, classe 456
  resumo do método 660
`java.lang.Comparable`, interface
  e classificação de dados reais 594–595
  parametrizada 595–596
  resumo do método 660
`java.lang.Double`, classe
  resumo do método 661
`java.lang.Error`, classe
  classe e hierarquia de exceção 456
  resumo do método 661
`java.lang.Exception`, classe
  classe e hierarquia de exceção 456
  exceções verificadas 458
`java.lang.IllegalArgumentException`, classe 455, 456
  como exceções não-verificadas 458
  resumo do método 661
`java.lang.IllegalStateException`, classe 456
  resumo do método 661
`java.lang.IndexOutOfBoundsException`, classe 456
`java.lang.Integer`, classe
  resumo do método 661–662
`java.lang.MalformedURLException`, classe 456
`java.lang.Math`, classe
  constantes 151
  métodos matemáticos 158*t*
  resumo do método 662–664
`java.lang.NullPointerException`, classe 456
  como exceções não-verificadas 458
  resumo do método 664
`java.lang.NumberFormatException`, classe 456
  capturando 461, 462
  como exceções não-verificadas 458
  resumo do método 664
`java.lang.Object`, classe
  como superclasse cósmica 425–431
  resumo do método 664
`java.lang.RuntimeException`, classe 456
  exceções não-verificadas 458
  resumo do método 664
`java.lang.String`, classe 63–65
  resumo do método 665
`java.lang.System`, classe 51–52
  resumo do método 666
`java.lang.Throwable`, classe
  classe e hierarquia de exceção 456
  resumo do método 666
`java.lang.UnknownHostException`, classe 456
`java.math`, pacote 666–667
`java.math.BigDecimal`, classe
  números grandes 149
  resumo do método 666–667
`java.math.BigInteger`, classe
  números grandes 149
  resumo do método 667
`java.net`, pacote 339*t*
`java.sql`, pacote 339*t*
`java.util`, pacote 339*t*, 667–673
`java.util.ArrayList`, classe 621
  resumo do método 667–668
  utilizando 276–280
`java.util.Arrays`, classe
  classificando dados reais 593
  método de classificação 587
  método de pesquisa binária 593
  resumo do método 668
`java.util.Calendar`, classe
  resumo do método 668
`java.util.Collections`, classe
  método de classificação 594
  resumo do método 669
`java.util.EventObject`, classe
  resumo do método 669
`java.util.GregorianCalendar`, classe
  resumo do método 669
`java.util.HashMap`, classe
  resumo do método 670
`java.util.HashSet`, classe
  resumo do método 670
`java.util.InputMismatchException`, classe 456
  resumo do método 670

`java.util.Iterator`, classe
utilizando 606
`java.util.Iterator`, interface
resumo do método 670
`java.util.LinkedList`, classe 621
implementando 610–619
resumo do método 670
utilizando 605–607
`java.util.List`, interface
resumo do método 670
`java.util.ListIterator`, interface
resumo do método 671
utilizando 605–606, 611–612
`java.util.logging`, pacote 673–674
`java.util.logging.Level`, classe
resumo do método 673–674
`java.util.logging.Logger`, classe
resumo do método 673–674
utilizando 213
`java.util.NoSuchElementException`, classe 456
capturando 461, 462
resumo do método 671
`java.util.PriorityQueue`, classe
resumo do método 672
`java.util.Random`, classe
como classe de ator 316–317
gerador de número aleatório 246
resumo do método 672
`java.util.Scanner`, classe
como classe de ator 316–317
lendo e gravando arquivos de texto 452
lendo entrada 169–172
resumo do método 672–673
`java.util.Set`, interface
resumo do método 673
`java.util.TreeMap`, classe
resumo do método 673
`java.util.TreeSet`, classe
resumo do método 673
`javadoc`,utilitário 110–111
copia primeira frase de comentários para tabela de resumo 107–110
`javax.swing`, pacote 339*t*, 673–679
`javax.swing.AbstractButton`, classe
resumo do método 673–674
`javax.swing.border`, pacote 678–679
`javax.swing.border.EtchedBorder`, classe
resumo do método 678–679
`javax.swing.border.TitledBorder`, classe
resumo do método 678–679
`javax.swing.ButtonGroup`, classe
resumo do método 674–675
`javax.swing.event`, pacote 678–679
`javax.swing.event.ChangeEvent`, classe
resumo do método 678–679
`javax.swing.event.ChangeListener`, interface
resumo do método 678–679
`javax.swing.event.JTextComponent`, classe
resumo do método 679
`javax.swing.ImageIcon`, classe
resumo do método 674–675
`javax.swing.JButton`, classe
construindo aplicativos com 384
resumo do método 674–675
`javax.swing.JCheckBox`, classe
resumo do método 674–675
`javax.swing.JComboBox`, classe
resumo do método 674–675
`javax.swing.JComponent`, classe
desenhando formas em janelas de frame 81–85
resumo do método 675–676
`javax.swing.JFileChooser`, classe
resumo do método 675–676
`javax.swing.JFrame`, classe
adicionando método `main` a 433–434
frames personalizados 432–434
janelas de frame 79–82
resumo do método 675–676
`javax.swing.JLabel`, classe
resumo do método 676–677
`javax.swing.JMenu`, classe
resumo do método 676–677
`javax.swing.JMenuBar`, classe
resumo do método 676–677
`javax.swing.JMenuItem`, classe
resumo do método 676–677
`javax.swing.JOptionPane`, classe
resumo do método 676–677
`javax.swing.JPanel`, classe
contruindo aplicativos com botões 384
resumo do método 677–678
`javax.swing.JRadioButton`, classe
resumo do método 677–678
`javax.swing.JScrollPane`, classe
resumo do método 677–678
utilizando 437
`javax.swing.JSlider`, classe
resumo do método 677–678

javax.swing.JTextArea, classe
  resumo do método 677–678
  utilizando 437
javax.swing.JTextField, classe
  personalização de frame 433
  processando entrada de texto 433–436
  resumo do método 677–679
javax.swing.text, pacote 679
javax.swing.Timer, classe
  resumo do método 678–679
JButton, construtor 674–675
JCheckBox, construtor 674–675
JComboBox, construtor 674–675
JFileChoose, construtor 675–676
JLabel, contêiner 384, 676–677
JMenu, construtor 676–677
JMenuBar, construtor 676–677
JMenuItem, construtor 676–677
JPanel contêiner 384
JRadioButton, construtor 677–678
JScrollPane, construtor 677–678
JSlider, construtor 677–678
JSwat 248
JTextArea, construtor 677–678
JTextField, construtor 677–679

K

KeyPad.java 522–524
keySet, método
  interface java.util.Map 671
Koch, floco de neve de 569–568

L

laço e meio, um 238–239, 243, 245
laços
  Como Fazer 242–244
  depurando 257
  for 231–237
  infinitos 229
  limites simétrico e assimétrico 244
  não utilizar != para testar o fim de intervalo 237
  processando valores sentinela 237–241
  provas de correção 248
  while 226–230
laços aninhados 237–238
  arrays bidimensionais 290
lançando exceções 455–457
Latin-1, subconjunto do Unicode 645*t*
leiaute
  forma livre 44
  instável 156, 642
leiaute de chaves 189–190
length, campo, arrays 274
length, método
  classe java.io.RandomAccessFile 659
  classe java.lang.String 63–65, 665
LIGHTGRAY, valor RGB 87–88*t*
limites assimétricos 244
limites simétricos 244
Line2D.Double, construtor 654
LinearSearchDemo.java 589–590
LinearSearcher.java 589
LineItem.java 501
LineNumberer.java 453–454
linguagens de criação de scripts 431
linguagens de programação 388
linguagens de programação de alto nível 36–38
linguagens de programação funcional 325
linhas
  Como Fazer 131, 132
  desenho 85–87
linhas em branco 641
LinkedList.java 615–618
Linux
  programas Java executarão em 38–39
listas de arrays 276–281
  Como Fazer 293
  contando correspondências 285–286
  localizando o máximo ou o mínimo 286–287
  localizando um valor 286
  número de elementos 281*t*
  parametrizados 281
  visão abstrata e concreta de 619–621
listas encadeadas
  circulares 629
  duplamente encadeadas 607
  eficiência de operações 622*t*
  implementando 609–620
  utilizando 604–608
  visão abstrata e concreta de 619–621
listIterator, método
  interface java.util.List 606, 670
ListIterator.java 618–619
ListTester.java 607–608
log, método
  classe java.lang.Math 158*t*, 663

`long` tipo 147*t*
classe empacotadora 282*t*
`Long`, classe 282*t*
`LoopFib.java` 552–553
Lovelace, Ada Augusta, condessa 588

M

Macintosh OS
programas Java executarão em 38–39
`MAGENTA`, valor RGB 87–88*t*
`main`, método 46
classes com 316–317
mainframes 79–80
Máquina Virtual Java 36, 38–39
e processo de compilação 51–53
máquinas eletrônicas de votação 126–127
Mark II, primeiro bug localizado no 258
`max`, método
classe `java.lang.Math` 158*t*, 663
`Measurer.java` 374
memória 32–33
memória de acesso randômico (random-access memory-RAM) 32–35
mensagens de rastreamento 213, 549
`MergeSortDemo.java` 584
`MergeSorter.java` 582–584
metáfora para programação 483
método da ficha CRC 486–487
e diagramas UML 490–491
estudo de caso da impressão de fatura 492–495
estudo de caso do caixa automático 505–508
método de superclasse 410
falhando ao invocar 412
métodos 46, 63–65
classe e descoberta de, no processo de projeto 486
com número variável de parâmetros 299
comentando 107–109
construtores, comparados com 71–72
controle de acesso 422–425
em diagramas UML 491
exceção 457
guia de estilo 637–638
guia de estilo requer um comentário para cada 636
herdados, tornando menos acessíveis 424
implementando 112–119
interface pública de classe 102–108
máximo de 30 linhas de código 636, 638
modificando assinatura na implementação 380–381
parâmetros implícitos e explícitos 123–127
pós-condições 328–329
pré-condições 326–328
subclasses 401
terminando ao lançar
valores constantes em 150
vinculação inicial e tardia 370
Ver também parâmetros; valores de retorno
métodos abstratos 422
métodos auxiliares recursivos 546–548
métodos de acesso 71–73, 321
clonando campos de instância mutáveis em 431
métodos de classe 329
métodos de instância 329, 637
polimórficos 369
métodos de predicado 206
métodos estáticos 329–331, 637
chamando 161–162
`main` sempre dever ser 46
nenhum parâmetro implícito 124–125
métodos matemáticos 158*t*
métodos modificáveis 71–73, 321
métodos recursivos 535–536
rastreando 542
`min`, método
classe `java.lang.Math` 158*t*, 663
ML 325
`mod`, método
classe `java.math.BigInteger` 667
modelo de cascata, desenvolvimento de software 481–482
modelo em espiral, desenvolvimento de software 482–483
monitor de computador 34–35
mouse 34–35
atalhos pelo teclado para operações 198–199
`mouseClicked`, método
interface `java.awt.event.MouseListener` 653
`mouseEntered`, método
interface `java.awt.event.MouseListener` 653
`mouseExited`, método
`java.awt.event.MouseListener` interface 653

`mousePressed`, método
  interface `java.awt.event.MouseListener` 653
`mouseReleased`, método
  interface `java.awt.event.MouseListener` 653
`MoveTester.java` 73–74
multiplicação 156
multiplicidades 491
`multiply`, método
  classe `java.math.BigDecimal` 667
  classe `java.math.BigInteger` 667

N

n-ésimo número triangular 534
`new`, operador 70–71
`next`, método
  classe `java.util.Scanner` 170, 452, 461, 673
  interface `java.util.Iterator` 606, 611–613, 670
`nextDouble`, método
  classe `java.util.Random` 246, 672
  classe `java.util.Scanner` 170, 452, 673
`nextInt`, método
  classe `java.util.Random` 246, 672
  classe `java.util.Scanner` 170, 206, 452, 458, 673
`nextLine`, método
  classe `java.util.Scanner` 170, 452, 673
nodos 604, 609–611
nome sobrecarregado 67–68, 369
nomes de arquivo 454
nomes de classe 62
  guia de estilo para 636, 641
nomes de método 62, 103–104
  guia de estilo 636, 640
  sobrecarregando 67–68, 369
nomes de variável 60–62
  guia de estilo 636, 640, 641
  utilização descritiva 154
notação camelo 62
notação científica 68–69
notação exponencial 68–69
nova linha, Unicode para 643*t*
números 68–69
  formatando 172
  grandes 149
  objetos, comparados com 78–79
números aleatórios 245–247
números binários 149
números de linha 453
números de ponto flutuante 68–69
  comparando 191–192
números mágicos, proibidos pelo guia de estilo 154, 636, 638
números pseudoaleatórios 246
números triangulares 534–537

O

O grande (big-Oh), notação 580, 581, 622
`ObjectInputStream`, construtor 657
`ObjectOutputStream`, construtor 657
  esquecendo de inicializar 123–124
  referências de objeto 77–80
objetos 46, 59, 63–65
  comparando 193–195
  construindo 69–72
  números, comparados com 78–79
  tipos e variáveis 60–62
ômega, notação 581
`omg.org.CORBA` 339*t*
operações aritméticas 156–161
operador de igual 191*t*
operador de incremento (++) 155–156
operador diferente de 191*t*
operador lógico *não* 206, 207
  e lei de De Morgan 209
operador maior que 191*t*
operador maior que ou igual 191*t*
operador menor que 191*t*
operador menor que ou igual 191*t*
operadores aritméticos 68–69
  combinando com atribuição 156
operadores binários 636, 641
operadores booleanos 206–207
  avaliação preguiçosa 209
  lei de De Morgan 209
operadores lógicos 206–207
operadores relacionais 191*t*, 191–192
  utilizando múltiplos 207–208
`ORANGE`, valor RGB 87–88*t*
ordenação total, relacionamento de 594
ou, operador lógico 206, 207
  avaliação preguiçosa 209
  confundindo com operador *e* 207–208
`OutOfMemory`, erro 458
ouvintes de evento 377–381, 439–440
  esquecendo de anexar 387
  não utilizar contêineres como 387–388
overflow, de cálculos 146–147

P
pacote de armazenamento em banco de dados 39
pacote de criptografia 39
pacote de rede 39
pacote de som 39
pacote gráfico 39
pacote padrão 340
pacotes da biblioteca, *Ver* Java, pacotes
pacotes, *Ver* Java, pacotes
padrões de codificação 484
padronização 626
painéis 384
`paintComponent`, método
  classe `javax.swing.JComponent` 82–85, 128–129, 133–134, 675–676
palavras reservadas 61
palavras-chave
  espaços em torno de 636
  somente em caixa baixa 50–51
parâmetros 46, 47, 103–105
  comentando 108–109
  e valores de retorno 65–68
  guia de estilo para nomear 637
  implícitos e explícitos 123–127
  imprimindo em rastreamento de programa 213
  métodos com número variável de 299
  tentando modificar tipos primitivos 323–324
parâmetros de construção 70–71
parênteses 641
  desbalanceados 160
  para multiplicação 68–69
`parseDouble`, método
  classe `java.lang.Double` 167, 661
`parseInt`, método
  classe `java.lang.Integer` 661
pastas 40–42
pastas aninhadas 40–41
pensando recursivamente 543–546
Pentium 4, chip 32–33
Pentium, bug de ponto flutuante do 149
perda de informações 148
permutações 538–542
permutando, na classificação 572–573
`PermutationGenerator.java` 540–541
`PermutationGeneratorDemo.java` 538–540
pesquisa binária 590–593
  acesso aleatório 604
pesquisa e substituição global 338
pesquisa linear 588
pesquisa seqüencial 588
pesquisando 588–590
  Ver também pesquisa binária; classificando
`PI`, constante, classe `java.lang.Math` 151
pilha de chamada 542
pilhas 623–625
`PINK`, valor RGB 87–88*t*
placa de rede 35
placa de som 35
placa gráfica 35
placa-mãe 34–35
planejamento realista 483
`Point2D.Double`, construtor 654
polígonos, desenhando 82–83
polimorfismo 368–370
  e herança 417–421
  e reutilização 359
ponto de interrupção, depurador 249–250
ponto, símbolo 343
ponto-e-vírgulas 46
  esquecendo, em laços 236
  guia de estilo 641
  muitos em laços 236
  omitindo 48–49
popriedade coletiva 484
por um, erros 229–230
  verificando 243
portabilidade, benefício do Java 37–39
portas 35
pós-condições 328–329
`pow`, método
  classe `java.lang.Math` 157, 158*t*, 663
precedência de operadores 631–633
pré-condições 326–328
`previous`, método
  interface `java.util.ListIterator` 607, 671
primeiro a entrar, primeiro a sair (first in, first out – FIFO) 623
`print`, método
  classe `java.io.PrintStream` 48, 63–64, 657
  classe `java.io.PrintWriter` 452, 658
`printf`, método
  classe `java.io.PrintStream` 658
  e formatação de números 172
`println`, método
  classe `java.io.PrintStream` 47, 51–52, 63–64, 657
  classe `java.io.PrintWriter` 452, 658

inspecionando estado atual de objetos 427
passando parâmetro para 65–66
`printStackTrace`, método
classe `java.lang.Throwable` 462, 666
`PrintWriter`, construtor 658
`PriorityQueue`, construtor 672
`private`, modificador 44
controle de acesso 422
todos os recursos na classe devem ser rotulados como públicos ou privados 637
processo formal de software
desenvolvimento 480
`Product.java` 502
produtividade de programadores 484
programa executável 636
programação 30–31
com pacotes 344–345
crie uma agenda e reserve tempo para problemas inesperados 203–204
defensiva 49–50
programação em par 484
Programação Extrema (Brock) 483
Programação Extrema 483–484
programadores
primeiros 588
produtividade 484
programadores de aplicativo 75–76
programadores de sistemas 75–76
programas de computador 30–31
compilando simples 43–48
traduzindo em código de máquina 36–38
Ver também erros
projeto orientado a objetos 100–103
ciclo de vida de software 480–484
descoberta de classe 485–487
estudo de caso: caixa automático 503–524
estudo de caso: imprimindo faturas 491–503
prompt 170
protótipos 482–483
provas de correção 248
`public`, modificador 44, 64–65, 105–106
coesão 318
comentando 107–110, 116–117
controle de acesso 422
esquecendo de definir métodos de implementação como públicos 366
projetando 102–108, 116–117
todos os recursos na classe devem ser rotulados como públicos ou privados 637
`put`, método
interface `java.util.Map` 671

Q

quadrado, Como Fazer 131
questões léxicas, guia de estilo 640–642
quicksort, algoritmo 588

R

RAM (Random-Access Memory) 32–35
`Random`, construtor 672
`Random`, objeto 316–317
`RandomAccessFile`, método
classe `java.io.RandomAccessFile` 659
rastreamento de programa 212-213
Rational Unified Process, método 483
`read`, método
classe `java.io.InputStream` 656
classe `java.io.Reader` 659
`readChar`, método
classe `java.io.RandomAccessFile` 659
`readDouble`, método
classe `java.io.RandomAccessFile` 659
`readInt`, método
classe `java.io.RandomAccessFile` 659
`readObject`, método
classe `java.io.ObjectInputStream` 657
`Rectangle`, construtor 652
`RectangleComponent.java` 83–84
`RectangleMeasurer.java` 374
`RectangleViewer.java` 83–85
recuo
e tabulações 190–191
guia de estilo 641
recuperação de falha de programa
e lançamento de exceções 455
recursão 533
Como Fazer 543–546
eficiência 548–554
infinita 538
mútua 554–560
números triangulares 534–537
permutações 538–542
`RecursiveFib.java` 548–550
`RecursiveFibTracer.java` 550–551
`RED`, valor RGB 87–88*t*
rede local (*local area network* – LAN)
salvando backups em 42
redes 32–35
redirecionamento de entrada 301

redirecionamento de saída 302
refatorando 484
referência a array 273
referência nula 194–196
referência para objeto 77–80
referências a subclasse 414
referências a superclasse 414
registrando em log 212–213
relacionamento *é-um* 364, 488
  Ver também herança
relacionamento *tem-um* 488
  Ver também agregação
relacionamentos
  classe entre 485, 488–491
  símbolos de relacionamento UML 490*t*
  visualização 319
relatórios de exceções 168–169
`remove`, método
  classe `java.util.ArrayList` 278, 668
  classe `java.util.PriorityQueuue` 672
  interface `java.util.Collection` 669
  interface `java.util.Iterator` 607, 612, 670
  interface `java.util.Map` 671
`removeFirst`, método
  classe `java.util.LinkedList` 605, 612, 670
`removeLast`, método
  classe `java.util.LinkedList` 605, 670
`repaint`, método
  classe `java.awt.Component` 650
  esquecendo de repintar 388
`replace`, método
  classe `java.lang.String` 66–68, 665
responsabilidades
  das classes 485, 486
  no método de fichas CRC 486–487
retângulos
  Como Fazer 131–134
  construindo 69–71
  desenhando 82–83
retorno de chamada
  método `actionPerformed` como 379
  utilizando interface para 370–374
`return`, comando 114–115
  Unicode para 643*t*
reutilização 441
  e herança 402
  e polimorfismo 359
  interfaces para 360–365
rotina de tratamento de exceção 457
rótulos 384
`round`, método
  classe `java.lang.Math` 148, 158*t*, 162, 663

S

saída 46–47
  exibindo a partir da caixa de diálogo 172
saída do sistema 46–47
salvando 42
`SavingsAccount.java` 421
`Scanner` objeto 316–317
`Scanner`, construtor 672
Scheme 325
scripts de shell 302
`seek`, método
  classe `java.io.RandomAccessFile` 659
segmentos de linha, desenhando 82–83
segurança, benefício de Java 37–39
seleção, operador de 191
`SelectionSortDemo.java` 574
`SelectionSorter.java` 573–574
`SelectionSortTimer.java` 577–579
seqüências de escape 169
`set`, método
  classe `java.util.ArrayList` 276–277, 622, 668
  interface `java.util.ListIterator`, 613–614, 671
`setBorder`, método
  classe `javax.swing.JComponent` 675–676
`setColor`, método
  classe `java.awt.Graphics` 92–93, 651
`setDefaultCloseOperation`, método
  classe `javax.swing.JFrame` 79–81, 675–676
`setEditable`, método
  classe `javax.swing.JComboBox` 674–675
  classe `javax.swing.JTextComponent` 437, 679
`setFont`, método
  classe `javax.swing.JComponent` 675–676
`setJMenuBar`, método
  classe `javax.swing.JFrame` 675–676
`setLayout`, método
  classe `java.awt.Container` 650
`setLevel`, método
  classe `java.util.logging.Logger` 213, 673–674
`setLine`, método
  classe `java.awt.geom.Line2D` 654
`setLocation`, método
  classe `java.awt.geom.Point2D` 654
  classe `java.awt.Rectangle` 652

setPreferredSize, método
classe java.awt.Component 441, 650
setSelected, método
classe javax.swing.AbstractButton 673–674
setSize, método
classe java.awt.Component 650
classe java.awt.Frame 79–80
setText, método
classe javax.swing.JTextComponent, 437, 679
setTitle, método
classe java.awt.Frame 79–80, 651
setVisible, método
classe java.awt.Component 650
Short, classe 282*t*
short, tipo 147*t*
classe empacotadora 282*t*
showInputDialog, método
classe javax.swing.JOptionPane 320, 676–677
showMessageDialog, método
classe javax.swing.JOptionPane 320, 676–677
showOpenDialog, método
classe javax.swing.JFileChooser 675–676
showSaveDialog, método
classe javax.swing.JFileChooser 675–676
silenciando exceções, guia de estilo recomenda usar especificação throws 640
símbolo indefinido, erro de 50–51
simplicidade 483
simulação 245–247
sin, método
classe java.lang.Math 158*t*, 663
sintaxe
acesso a elemento do array 275
assertiva 327
atribuição 63
bloco de instrução 189
bloco try geral 461
chamada de método 48
chamada de método estático 162
chamando construtor da superclasse 413
chamando método da superclasse 411
classes internas 375
cláusula finally 464
coerção 148
comando if 188
construção de array 274
construção de objeto 70–71
declaração de campo de instância 111–112
definição de classe 106–107
definição de constante 151
definição de construtor 106–107
definição de interface 364
definição de método 106–107
definição de variável 61
especificação de exceção 460
especificação de pacote 340
herança 403
implementação de interface 364
importando uma classe de um pacote 74–75
instrução for 231
instrução return 114–115
instrução while 228
laço for each 285
lançando exceção 457
operador instanceof 416
sistema de arquivos 42–43
Sistema de Declaração de Imposto de Renda Norte-Americano 200*t*
sistemas operacionais 377
size, método
classe java.util.ArrayList 276–277, 668
interface java.util.Collection 669
solução recursiva 535
sombreando 337–338
campos de instância 411–412
sort, método
classe java.util.Arrays 587, 593, 668
classe java.util.Collections 594, 669
sqrt, método
classe java.lang.Math 157, 158*t*, 162, 663
como método estático 330
Stack, classe 624
start, método
classe java.applet.Applet 649
classe javax.swing.Timer 678–679
stateChanged, evento 678–679
step into (entrar), comando 250, 251
step over (pular), comando 250, 251
stop, método
classe java.applet.Applet 649
classe javax.swing.Timer 678–679
StopWatch.java 576–577
StringIndexOutOfBoundsException 168
strings 47, 166–169
comparando 192–194
e tipo char 169
número de elementos 281*t*

strings vazias 166
Structured Query Language (SQL) 339*t*
subclasses 401
  confundindo com superclasses 404
  construção 413–414
  convertendo em tipo de superclasse 414–416
  herdando campos de instância e métodos 407–412
subconjunto 404
`substring`, método
  classe `java.lang.String` 167, 665
subtração 156
`subtract`, método
  classe `java.math.BigDecimal` 667
  classe `java.math.BigInteger` 667
Sun Microsystems 37–39
`super`, palavra-chave 410, 412, 413
superclasse, construtor de 413
superclasses 401
  classe `java.lang.Object` como "cósmica" 425–431
  confundindo com subclasses 404
  convertendo em tipo de subclasse 414–416
  herdando campos de instância e métodos 407–412
superconjunto 404
`SwingConstants`, interface 366
`switch`, instrução 198–199
`SyllableCounter.java` 252–253
`System.in`, objeto 169–170
`System.out`, objeto 47, 51–52, 63–65, 169–170

T

tabela de resumo 108–110
tabulações
  e indentação 190–191
  guia de estilo para configurar 636, 641
  Unicode para 643*t*
`tan`, método
  classe `java.lang.Math` 158*t*, 663
`TaxCalculator.java` 202
`TaxReturn.java` 200–202
teclado 34–35
  atalhos para operações de mouse 198–199
tempo de execução, erro de 49–50, 52–53
tempo de execução, pilha de 624
teste da caixa preta 210
teste de igualdade 191
testes
  cálculos 165
  classes 118–119
  cobertura do teste 210-213
  criando agenda e reservando tempo para problemas inesperados 203–204
  de regressão 300–302
  de unidade 118–121
  de unidade com ferramenta JUnit 345–347
  e Programação Extrema 484
  implementando programa de teste 72–75
  interativos 74–75, 119–120
  para referência nula 194–196
texto
  Como Fazer 132
  lendo e gravando arquivos 452–455
  processando eventos de entrada 433–436
Therac-25, incidentes com 302
theta, notação 581
`this`, referência 126–127
  para sombreamento 337
`throw`, instrução 457
`Throwable`, construtor 666
throws 327
  guia de estilo para utilizar 640
`throws`, especificador 459
`TicTacToe.java` 290–291
`TicTacToeRunner.java` 291–292
`Timer`, construtor 678–679
timers 388
tipo abstrato de dados 619–622
tipo concreto de dados 619–622
tipo de classe
  convertendo em tipo de interface 367–368
tipo de retorno 103–105
tipo variável 60–62
tipo, parâmetro 276–277
tipos 60–62
tipos de interface 360–364
  convertendo em classe 367–368
tipos enumerados 205, 431
tipos numéricos 68–69, 146–149
tipos primitivos 68–69, 100–101, 146, 147*t*
  empacotadores e auto-empacotamento 281–282
  tentando modificar 323–324
`TitledBorder`, construtor 678–679
`toDegrees`, método
  classe `java.lang.Math` 158*t*, 664

`toLowerCase`, método
classe `java.lang.String` 665
`toRadians`, método
classe `java.lang.Math` 158*t*, 664
`toString`, método
classe `java.lang.Enum` 431
classe `java.lang.Integer` 662
classe `java.lang.Object` 401, 425*t*, 452, 664
e herança 427
fornecido para todas as classes 427
sobrescrevendo 426–427
`toUpperCase`, método
classe `java.lang.String` 64–65, 321, 665
transistores 32–33
`translate`, método
classe `java.awt.Rectangle` 71–73, 76–77, 79, 652
tratamento de exceções 455
abordagem lançar cedo, capturar tarde 462
capturando exceções 460–463
cláusula `finally` 463–465
exceções verificadas e não-verificadas 458–460
exemplo de estudo de caso 467–471
lançando exceções 455–457
`TreeMap`, construtor 673
`TreeSet`, construtor 673
`Triangle.java` 536–537
`TriangleTester.java` 537
`true`, `boolean` tipo 205
`try`, bloco 460–463
e cláusula `finally` 464
`try/catch`, instrução 460–463

U

último a entrar, primeiro a sair (last in, first out – LIFO) 623, 624
UML (Unified Modeling Language) 319
UML, diagramas
atributos e métodos em 491
e ficha CRC 490–491
estudo de caso da impressão de fatura 495
estudo de caso do caixa automático 508–509
UML, notação 489
UML, símbolos de relacionamento 490*t*
Unicode 169
caracteres de controle selecionados 643*t*
subconjunto Latino básico (ASCII), 644*t*
subconjunto Latino-1 645*t*
unidade central de processamento (*central processing unit* – CPU) 32–35
`union`, método
classe `java.awt.Rectangle` 652
UNIX
caminho de classe 342
desempenho com conjuntos de dados grandes ou aleatórios 298
programas Java executarão em 38–39
separador de arquivo 345
"usa", relacionamento 364, 489
uso de letras maiúsculas e minúsculas 44
uso de letras maiúsculas e minúsculas, verificação 50–51

V

valores de retorno 65–68
comentando 108–109
imprimindo no rastreamento de programa 213
testando 119–120
valores RGB de cores 87–88*t*
valores sentinela 237–241
variáveis 60–62
acessando vizinhas 381–382
categorias de 121–124
compilador atribui posição na memória 37–38
guia de estilo 638–639
inspecionando com depurador 250
operador de atribuição 62–63
tempo de vida 121–123
todas não-final devem ser privadas 637
variáveis booleanas
quando a condição de terminação de laço é avaliada no meio do laço 238–239
utilizando 209–210
variáveis de array
guia de estilo 639
variáveis de instância 121–122
variáveis de interface 368
variáveis de objeto 60
múltiplas podem conter referências ao mesmo objeto 78
variáveis de parâmetro 121–124
não altere o conteúdo 325
variáveis estáticas
evitar sempre que possível 637
variáveis locais 121–124
acessando vizinhas 382

escopo 334–335
guia de estilo para nomear 641
variáveis numéricas 60
variável de índice 284
variável elemento 284
variável final 150, 382
verificação ortográfica 50–51
versões pequenas 483
vinculação inicial 370
vinculação tardia 370
vírgula 641

W

`while`, instrução 226–227
utilizando espaços em torno da parte (. . .) 641
`while`, laços 226–230
`break` e `continue` 245
Como Fazer 242–243
considerando o uso no lugar de laços `for` 235
e código espaguete 230
`WHITE`, valor RGB 87–88*t*
Windows XP 203–204
Windows, sistema operacional
caminho de classe 342
programas Java executarão em 38–39
separador de arquivo 345
`Word.java` 251–253
worm da Internet 300
`write`, método
classe `java.io.OutputStream` 657
classe `java.io.Writer` 659
`writeChar`, método
classe `java.io.RandomAccessFile` 659
`writeChars`, método
classe `java.io.RandomAccessFile` 659
`writeDouble`, método
classe `java.io.RandomAccessFile` 659
`writeInt`, método
classe `java.io.RandomAccessFile` 659
`writeObject`, método
classe `java.io.ObjectOutputStream` 657

Y

`YELLOW`, valor RGB 87–88*t*

Créditos das Ilustrações

⊕ Indica que a imagem está localizada em conteúdo online disponível no site da Bookman Editora.

Capítulo 1

Página 32: Copyright © 2007, Intel Corporation.
Página 33 (acima): PhotoDisc, Inc./Getty Images.
Página 33 (abaixo): PhotoDisc, Inc./Getty Images.
Página 34: Copyright © 2007, Intel Corporation.
⊕ Fato Aleatório 1.1 (O ENIAC): Cortesia de Sperry Univac, Divisão da Sperry Corporation.

Capítulo 2

Fato Aleatório 2.1 (Um computador mainframe): Corbis Digital Stock.

Capítulo 3

⊕ Fato Aleatório 3.1 (Eleições com cartão perfurado): David Young-Wolff/PhotoEdit.
⊕ Fato Aleatório 3.1 (Máquina de votar com tela sensível ao toque): Bob Daemmrich/The Image Works.
Página 131: Punchstock.
⊕ Fato Aleatório 3.2 (Diagramas): Copyright © 2001-2007 Lev Givon. Todos os direitos reservados.
⊕ Fato Aleatório 3.2 (Cena): Keith Kapple/SUPERSTOCK.
⊕ Fato Aleatório 3.2 (Imagem manipulada): Daniel Biggs/SUPERSTOCK.

Capítulo 5

Página 205: © 2007 Sidney Harris.

Capítulo 6

⊕ Fato Aleatório 6.3: Naval Surface Weapons Center, Dahlgren, VA.

Capítulo 8

⊕ Fato Aleatório 8.1: Captura de tela do VisiCalc © IBM Corporation. Utilizada com permissão.

Capítulo 11

Fato Aleatório 11.1 (esquerda, centro, direita): © AP/Wide World Photos.

Capítulo 12

Página 483: Booch/Rumbaugh/Jacobson, *Unified Modeling Language Users Guide*, Figure C-1 "The Software Development Life Cycle," © 2005, 1999 Pearson Education, Inc. Reproduzido com permissão da Pearson Education, Inc. Todos os direitos reservados.

Capítulo 13

Fato Aleatório 13.1 (Alan Turing): Science Photo Library/Photo Researchers.

Capítulo 14

Fato Aleatório 14.1: Topham/The Image Works.

Capítulo 15

Página 624: Photodisc/Punchstock.

Capítulo 16

Fato Aleatório 16.1: Courtesia de Nigel Tout.