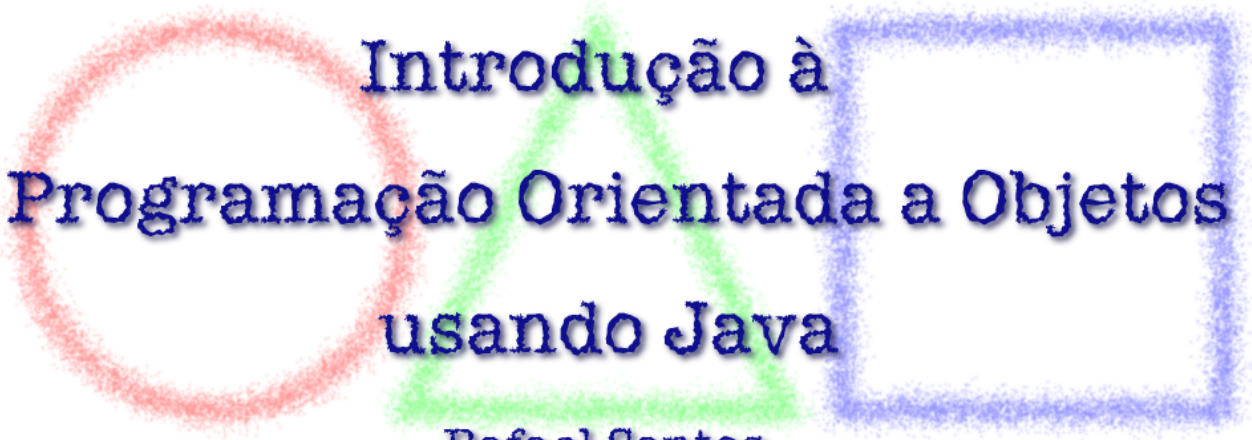


# Introdução à Programação Orientada a Objetos usando Java

Rafael Santos

# Prefácio

## Sobre este livro

Este livro tem como finalidade possibilitar ao estudante o aprendizado dos conceitos básicos de programação orientada a objetos usando a linguagem Java. O livro parte do pressuposto de que o estudante conhece e usa computadores, tem noções elementares de algoritmos e sabe o que é uma linguagem de programação, mas sem necessariamente possuir conhecimentos profundos de uma. Evidentemente o livro também pode ser usado por estudantes que já conheçam alguma linguagem de programação mas estejam interessados em aprender Java.

O livro foi inicialmente desenvolvido como sendo o material sugerido para as disciplinas *Programação Orientada a Objetos I* e *Programação Orientada a Objetos II* dos cursos de Ciência da Computação e Engenharia da Computação oferecidos pela Faculdade de Ciência da Computação da Universidade do Vale do Paraíba (UNIVAP). O material original foi ampliado consideravelmente para a publicação como livro.

Ao desenvolver este livro, minha motivação principal foi apresentar os conceitos de programação orientada a objetos com clareza e simplicidade, mostrando exemplos e sugerindo problemas práticos e ilustrativos dos conceitos e de algoritmos em geral. Muitos livros de programação orientada a objetos (abreviadamente, POO), especialmente usando a linguagem Java, concentram-se em aspectos visuais, deixando de lado a teoria e explicação dos detalhes que são necessárias para melhor compreensão das técnicas e conceitos. Na maioria dos casos, demonstrações das capacidades da linguagem são apresentadas desde cedo, sem as explicações necessárias para compreensão do que acontece nos programas-exemplo. Em contraste, este livro concentra-se nas técnicas de programação orientada a objetos, explicando a teoria e apresentando a prática de cada tópico. Espero que com isto os estudantes estejam mais confortáveis com os conceitos de classes e objetos quando forem utilizá-los em aplicações mais complexas.

Para reforçar a aplicabilidade deste livro em cursos de programação em escolas técnicas e universidades, **169** programas, classes e trechos de código foram incluidos, com comentários para esclarecer os pontos mais importantes. Ao final de cada capítulo, vários exercícios de fixação com diferentes níveis de complexidade são apresentados, em um total de **881** exercícios.

## Sobre a organização deste livro

Este livro é dividido em cinco partes principais, cada uma subdividida em vários capítulos. Cada capítulo cobre um tema específico. As partes e capítulos são:

- **Parte I: Introdução à Programação Orientada a Objetos** apresenta os conceitos básicos de programação orientada a objetos. Esta parte é dividida nos seguintes capítulos:

  - **Capítulo 1: Introdução à Programação Orientada a Objetos** apresenta conceitos básicos de programação orientada a objetos, esclarecendo as diferenças e relações entre modelos, classes, objetos e instâncias. Neste capítulo, os conceitos apresentados são puramente teóricos, mas **essenciais** para a compreensão dos princípios básicos de orientação a objetos.
  - **Capítulo 2: Criando Classes em Java** mostra como classes em Java são declaradas, e apresenta os tipos de dados e os mecanismos de criação de campos e métodos nas classes.
  - **Capítulo 3: Criando Aplicações em Java** mostra como podemos criar aplicações ou programas em Java (classes que podem ser executadas pela máquina virtual Java), e como podemos usar classes criadas anteriormente nestes programas.
  - **Capítulo 4: Construtores e Sobrecarga** apresenta *construtores*, métodos especiais que serão chamados quando objetos das classes que os contém forem instanciados. O capítulo também descreve como fazer a sobrecarga de métodos, que permite que vários métodos com o mesmo nome mas diferentes funções possam ser declarados e usados.
  - **Capítulo 5: Campos e Métodos Estáticos** mostra como podemos criar métodos e campos estáticos em aplicações ou outras classes em Java. Métodos e campos estáticos não precisam ser instanciados para serem usados.
  - **Capítulo 6: Estruturas de Decisão e Controle - Condicionais** apresenta as estruturas de decisão baseadas em condições, juntamente com uma introdução aos operadores lógicos de Java.
  - **Capítulo 7: Estruturas de Decisão e Controle - Repetição** apresenta as estruturas de repetição de Java baseadas em condições, juntamente com noções sobre recursão e otimização.
  - **Capítulo 8: Reutilização de Classes** mostra dois mecanismo de Programação Orientada a Objetos que permitem a uso de uma classe já existente para criação de uma classe derivada, mais especializada.
  - **Capítulo 9: Classes Abstratas e Interfaces** apresenta os conceitos de *classes abstratas* e *interfaces*, que também servem como mecanismos de herança de Java com características especiais.
  - **Capítulo 10: Pacotes (*packages*) de Classes em Java** mostra como classes podem ser organizadas em grupos chamados *pacotes*, e ilustra a criação e uso destes pacotes.

- **Parte II: Classes de Java para Manipulação de Dados** apresenta algumas classes para manipulação de dados que são padrão de Java. Esta parte é dividida nos seguintes capítulos:
  - **Capítulo 11: Arrays em Java** apresenta os mecanismos de criação e uso de *arrays* (vetores de dados) unidimensionais e multidimensionais em Java, assim como técnicas de Programação Orientada a Objetos para a criação de classes que implementam arrays de forma robusta.
  - **Capítulo 12: Classes para Manipulação de Strings** apresenta e demonstra o uso de três classes para processamento de cadeias de caracteres: `String`, `StringBuffer` e `StringTokenizer`.
  - **Capítulo 13: Classes para Representação e Processamento de Valores Numéricos** apresenta a classe `Math` e seus métodos e constantes usados para operações matemáticas, e classes que encapsulam valores numéricos representáveis por tipos nativos.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

  - **Capítulo ??: Classes para Representação de Estruturas de Dados** apresenta algumas classes que implementam estruturas de dados simples em Java e suas aplicações.
  - **Capítulo ??: Exceções** apresenta os mecanismos de tratamento e criação de exceções em Java para processamento de erros.
  - **Capítulo ??: Entrada e Saída usando Arquivos** apresenta as classes em Java que permitem a abstração de acesso à arquivos de diversos tipos.
  - **Capítulo ??: Recebendo e Enviando Dados via Internet** apresenta um mecanismo simples de envio e recebimento de dados via Internet.
  - **Capítulo ??: Programação Cliente-Servidor com *Sockets*** mostra como podemos criar aplicações simples com o modelo cliente/servidor e a classe `Socket`.
  - **Capítulo ??: Linhas de Execução (*Threads*)** mostra como programas podem ser preparados para executar mais de uma tarefa simultaneamente através do uso de linhas de execução.

- **Parte III: Classes de Java para Programas com Interfaces Gráficas** apresenta as classes principais de Java que permitem a criação de programas com interfaces gráficas, e está dividida nos seguintes capítulos:
  - **Capítulo ??: Introdução à Programação com Interfaces Gráficas** apresenta os conceitos básicos de programação com interfaces gráficas, e introduz as classes `Frame` e `Canvas`.
  - **Capítulo ??: Introdução à Programação com Eventos** apresenta os conceitos básicos de programação com eventos, e sua relação com programas com interfaces gráficas, além de demonstrar usos de instâncias da classe `Button`.
  - **Capítulo ??: Usando Diferentes Layouts Gráficos** mostra como *layouts* podem ser usados para distribuir diversos componentes em programas com interfaces gráficas.
  - **Capítulo ??: Usando Campos de Texto** apresenta as classes `Label`, `TextField` e `TextArea`, para apresentação e entrada de texto em programas com interfaces gráficas.
  - **Capítulo ??: Usando Listas e Opções** apresenta classes cujas instâncias representam listas e opções para interação com usuários.
  - **Capítulo ??: Usando Menus** mostra como menus podem ser criados em programas com interfaces gráficas, e como funções do programa podem ser executados caso opções dos menus sejam escolhidas.
  - **Capítulo ??: Introdução à Programação de Applets** mostra como *Applets* (programas que podem ser carregados via Internet e mostrados em um navegador) podem ser criados com os conceitos mostrados anteriormente.

- **Parte IV: Projetos Sugeridos** apresenta três projetos sugeridos para leitores interessados. Estes projetos são mostrados como aplicações quase completas mas que podem ser melhoradas com sugestões apresentadas como exercícios. Os projetos são:
  - **Capítulo ??: Um Colorizador de Código Java** propõe a criação de um programa que analisa e anota códigos em Java criando arquivos HTML que contém o código colorizado.
  - **Capítulo ??: Interface Orientada a Objetos para o Software POV** propõe a criação de classes que modelam os componentes gráficos usados no software gratuito *POV* (*Persistence of Vision*).

  - **Capítulo ??: Uma Planilha Simples** apresenta as estruturas e conceitos para desenvolvimento de uma planilha de cálculos simples, e exercícios para fazer um aplicativo mais útil.

- **Parte V: Apêndices**
  - **Apêndice A: Tipos de Dados em Java** apresenta os tipos de dados nativos em Java com exemplos de usos corretos e incorretos.
  - **Apêndice B: A classe `Keyboard`** apresenta a classe `Keyboard`, que pode ser usada para facilitar a leitura de valores de tipos nativos e instâncias da classe `String` a partir do teclado, adicionando flexibilidade aos programas que não usam interfaces gráficas.
  - **Apêndice B.1: Usando o *Java Development Kit*** apresenta a ferramenta escolhida para desenvolvimento de programas, classes e *applets* em Java.
  - **Apêndice ??: Documentação de Classes e Programas em Java** mostra como podemos criar documentação para aplicativos e classes em Java através da documentação do código-fonte.

## Convenções usadas neste livro

Para destacar alguns conceitos apresentados neste livro, fontes diferentes serão usadas. Nomes de classes, métodos, pacotes, comandos, etc. serão mostrados em fonte `proporcional`. Conceitos apresentados pela primeira vez ou que mereçam destaque e palavras que não sejam parte do português corrente serão mostrados em fonte *itálica*.

Adicionalmente, os programas incluídos no livro serão *colorizados*, isto é, palavras-chave e partes do código como comentários serão mostradas em cores diferentes para facilitar a leitura. As linhas dos programas também serão numeradas para fácil referência.

*categorias de exercícios*

Os exercícios apresentados ao fim de cada capítulo são divididos em cinco categorias de dificuldade, marcadas com um número de estrelas correspondente:

- **Uma estrela (⋆)**: Exercícios teóricos ou práticos que podem ser resolvidos rapidamente, geralmente através de consultas a programas mostrados ou respostas de outros exercícios, e modificação de exemplos ou outras respostas. Em muitos casos podem ser resolvidos mentalmente, isto é, sem a necessidade de escrever, compilar e executar programas.
  Existem **330** exercícios desta categoria no livro.
- **Duas estrelas (⋆⋆)**: Exercícios teóricos ou práticos que exigem um pouco mais de raciocínio e modificações ligeiras de exemplos apresentados. Exercícios desta categoria geralmente precisarão ser compilados e executados para verificação dos resultados.
  Existem **229** exercícios desta categoria no livro.
- **Três estrelas (⋆⋆⋆)**: Exercícios que requerem compreensão mais completa dos conceitos envolvidos, mas mesmo assim podem ser resolvidos com base em exemplos ou respostas a outros exercícios.
  Existem **227** exercícios desta categoria no livro.
- **Quatro estrelas (⋆⋆⋆⋆)**: Exercícios que requerem compreensão ainda mais completa e profunda dos conceitos, e que geralmente exigem um conhecimento razoável de algoritmos e técnicas de programação.
  Existem **68** exercícios desta categoria no livro.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

- **Cinco estrelas (★★★★★)**: Exercícios que requerem a solução de um problema mais completo e complexo, envolvendo vários conceitos diferentes da disciplina. Estes exercícios podem servir como base para projetos mais interessantes.
  Existem **27** exercícios desta categoria no livro.

## Sobre o Site de Suporte

Material complementar a este livro, na forma de um *site de suporte* ao livro, pode ser encontrado na WWW acessando a URL http://www.bioinfo.univap.br/~POO.

Presentemente o site contém vários arquivos para *download* (classes úteis, arquivos usados nos exemplos mostrados no livro), as listagens dos programas do livro com sintaxe colorizada no formato PDF e *links* para softwares úteis para o desenvolvimento de programas em Java. Estão em desenvolvimento ferramentas que podem ser usadas pela Internet, como um colorizador de código via WWW e um compilador que indica erros apresentando mensagens em português com mais detalhes do que o próprio compilador presentemente incluído no kit de desenvolvimento Java.

## Por que Java ?

*por que usar Java no ensino de POO ?*

Existem diversas linguagens de programação orientadas a objeto, cada uma com diferentes características e apelos de mercado, educacionais ou acadêmicos. Nesta seção, algumas das razões da escolha da linguagem Java serão apresentadas.

**Java é *obrigatoriamente* orientada a objetos.** Algumas linguagens permitem que objetos e variáveis existam em diversos pontos de um programa, como se estivessem desatreladas de qualquer estrutura. Em Java, **todas** as variáveis e métodos devem estar localizados dentro de classes, forçando o uso de orientação a objetos até mesmo em tarefas simples. Dessa forma, o estudante de programação orientada a objetos que esteja usando Java estará usando mais as técnicas de POO.

**Java é simples.** A estrutura de programas e classes em Java segue a organização de linguagens tradicionais como C e C++, mas sem elementos que tornam programas e programação mais complexos. Após o aprendizado dos conceitos básicos de programação orientada a objetos, o estudante da linguagem pode começar a criar aplicativos úteis e complexos.
A simplicidade se reflete também na maneira com que arquivos contendo programas em Java são compilados e executados: se as recomendações básicas forem seguidas, o compilador se encarregará de compilar todas as classes necessárias em uma aplicação automaticamente, sem necessidade de arquivos adicionais de configuração e inclusão de bibliotecas.

**Java é portátil.** O código-fonte de um programa ou classe em Java pode ser compilado em qualquer computador, usando qualquer sistema operacional, contanto que este tenha uma máquina virtual Java adequada (veja o apêndice B.1). Adicionalmente, as classes criadas podem ser copiadas e executadas em qualquer computador nas mesmas condições, aumentando a utilidade da linguagem através da independência de plataformas, contanto que versões compatíveis da máquina virtual sejam usadas.

**Java é gratuita.** A máquina virtual Java, mencionada acima, está à disposição para cópia no site da Sun e em vários outros. Compiladores simples, de linha de comando (sem interfaces visuais elaboradas) fazem parte do *JDK*, o ambiente de desenvolvimento gratuito de Java. Aplicações em Java precisam de uma máquina virtual para sua execução, mas não



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

existem limitações na distribuição da máquina virtual, fazendo de Java uma plataforma extremamente econômica para desenvolvedores e usuários finais.

**Java é robusta.** Administração de memória (alocação e liberação) e o uso de ponteiros, duas das fontes de erros e *bugs* mais frequentes em programas em C e C++, são administrados internamente na linguagem, de forma transparente para o programador. De maneira geral, programas em Java tem restrições no acesso à memória que resultam em maior segurança para os programas sem diminuir a utilidade dos mesmos.
Java também tem um poderoso mecanismo de exceções que permite melhor tratamento de erros em tempo de execução dos programas.

**Java tem bibliotecas prontas para diversas aplicações.** As bibliotecas de classes de Java contém várias classes que implementam diversos mecanismos de entrada e saída, acesso à Internet, manipulação de Strings em alto nível, poderosas estruturas de dados, utilitários diversos e um conjunto completo de classes para implementação de interfaces gráficas.
Vale a pena relembrar que estas bibliotecas são padrão de Java - qualquer máquina virtual Java permite o uso destas bibliotecas, sem a necessidade de instalar pacotes adicionais, e que mesmo que o compilador usado não tenha interface gráfica similar à de linguagens visuais, os programas criados com este compilador podem ter interfaces gráficas complexas.

### Categorias de softwares em Java

Existem basicamente quatro categorias de softwares que podem ser criados em Java:

*classes para representação de modelos*

**Classes para representação de modelos.** Classes são usadas para representar, em linguagem de programação, modelos e abstrações de dados como os mostrados nos capítulos 1. Classes para representação de modelos não podem ser executadas diretamente, mas instâncias destas classes podem ser usadas dentro de aplicações e *applets*. Detalhes e mecanismos de criação de classes em Java serão vistos no capítulo 2.

*bibliotecas*

**Classes como conjuntos de rotinas.** Classes podem conter somente métodos (rotinas) ao invés de representar dados. Estas classes funcionam como *bibliotecas* de métodos ou funções que tem algo em comum. Detalhes sobre a criação deste tipo de classes serão vistos na seção 5.3.

*aplicações*

**Aplicações ou Programas.** Aplicações ou programas podem ser executadas a partir de um terminal do sistema operacional (terminais gráficos de Unix ou da janela do MS-DOS), podendo envolver ou não maior interação com o usuário, e podendo ou não ter uma interface gráfica. Informações sobre a criação de programas em Java serão mostradas no capítulo 3 (sem uso de interface gráfica) e nos capítulos da parte III (com interfaces gráficas).

*applets*

***Applets.*** *Applets* são programas executáveis em Java que podem ser embutidos em páginas em HTML e carregados e executados via Internet. Sua execução é controlada pelo navegador que está sendo usado para ler as páginas. Por razões de segurança, *applets* são limitadas nas tarefas que podem fazer, e obrigatoriamente devem ter interface gráfica. Detalhes e mecanismos de criação de *applets* em Java serão vistos no capítulo **??**.

## O que não será visto neste livro

Eis aqui uma lista dos tópicos que **não serão** cobertos pelo livro. Ao invés de considerar a lista como um atestado de que o livro está incompleto, por favor considere que o livro é uma introdução, e certos tópicos são complexos demais para um livro deste tipo ou simplesmente não são interessantes, úteis ou comuns o suficiente para merecer discussão.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

- Comandos `break` e `continue` com labels.
- *Inner classes* (classes aninhadas).

## Dedicatória e Agradecimentos

Este livro é dedicado a minha esposa Yukari, e a nossa filha Miyuki, pela infinita paciência, carinho e suporte demonstrados durante a criação deste livro; e aos meus pais pelas várias lições de vida.

Sou grato aos estudantes dos cursos da Faculdade de Ciência da Computação da Univap que contribuiram com idéias e perguntas que resultaram em exemplos e exercícios, e em especial, aos estudantes que participaram na revisão do livro. Entre estes, destaco Nádia Maria França Borges, por inúmeras correções e sugestões.

Sou também grato aos professores Paulo de Castro Lobo, Moacir de Almeida Prado e Carlos Manoel Fenile Péris por esclarecimentos necessários para alguns exercícios.

## Modificações

Esta seção somente será mantida enquanto o livro estiver em desenvolvimento.

### Plano de Modificações

A lista de modificações previstas para as próximas versões deste livro são, em ordem aproximada de prioridade:

1. Completar o capítulo 12.
2. Completar o capítulo 13.
3. Completar o apêndice B.1.
4. Completar o apêndice A.
5. Completar o apêndice B.
6. Escrever os capítulos restantes.

# Sumário





Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

# Lista de Figuras

# Lista de Listagens

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

# Lista de Tabelas

# Parte I

# Introdução à Programação Orientada a Objetos



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

# Capítulo 1

# Introdução à Programação Orientada a Objetos

## 1.1 O que é um programa de computador ?

Computadores são ferramentas de uso comum hoje em dia - praticamente qualquer atividade humana se beneficia ou pode se beneficiar do uso de computadores. Computadores podem automatizar tarefas como armazenar e manipular valores em contas bancárias ou cadastros de alunos, efetuar cálculos complexos ou repetidos com rapidez e confiabilidade, armazenar e procurar informações em grandes volumes de dados e também nos entreter com jogos e informações em multimídia.

*programas*

O que faz um computador ser capaz de efetuar estas tarefas são seus *softwares* ou seus *programas*. Os programas que um computador pode executar dizem a ele o que deve ser feito com seu *hardware* (seus componentes e periféricos): ler de e escrever em arquivos em um disco, exibir informações em um monitor, efetuar cálculos usando o processador, ler a posição do mouse ou caracteres do teclado, etc.

*linguagens de programação*

*código*

*compilador*

Programas são conjuntos de comandos e regras que um programador deve conhecer para poder manipular os recursos de um computador. Programas são escritos usando *linguagens de programação*, que definem regras específicas e bem determinadas e um conjunto de operadores e comandos que podem ser usados. O conteúdo dos programas, escrito por programadores de forma que outros programadores possam ler e entendê-los também é chamado de *código* ou *código-fonte*. Para que o código seja executado por um computador, ele deve ser traduzido da linguagem de programação (alto nível) para uma linguagem que possa ser compreendida pelo computador através de um *compilador*.

*dados*

Programas processam *dados*: valores em uma conta bancária, caracteres entrados por um teclado, pontos em uma imagem, valores numéricos para cálculos. O paradigma de Programação Orientada a Objetos considera que os dados a serem processados e os mecanismos de processamento destes dados devem ser considerados em conjunto. A criação de modelos que representam conjuntamente dados e operações nestes dados, é descrita na próxima seção.

## 1.2 O que são modelos ?

*modelos*

*Modelos* são representações simplificadas de objetos, pessoas, itens, tarefas, processos, conceitos, idéias, etc., usados comumente por pessoas no seu dia-a-dia, independente do uso de com-



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

putadores.

Para exemplificar o uso de modelos em tarefas comuns, consideremos o Restaurante Caseiro Hipotético, que serve refeições por quilo, e onde o gerente, que também é a pessoa que fica na balança e no caixa, anota os pesos dos pratos dos clientes e os pedidos que os garçons trazem em um quadro-branco. A figura 1.1 mostra o quadro-branco do Restaurante Caseiro Hipotético.

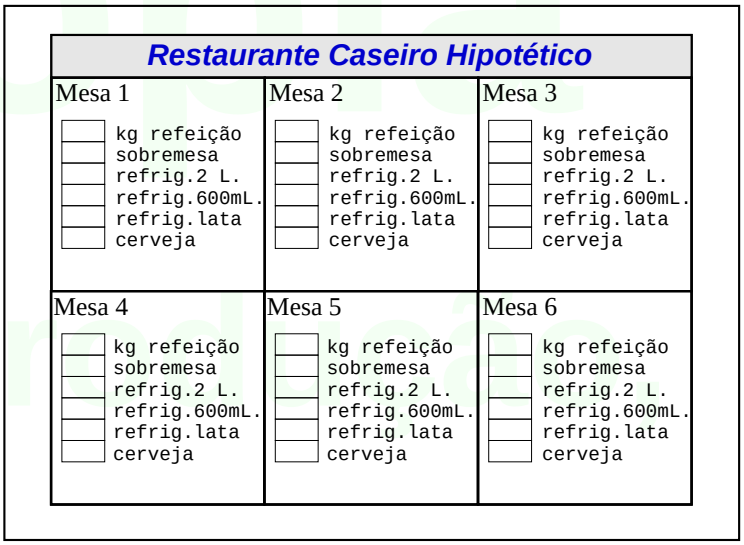



Figura 1.1: O quadro-branco do Restaurante Caseiro Hipotético

O quadro-branco mostrado na figura 1.1 contém espaços para anotar o peso dos pratos e os pedidos de cada mesa do restaurante. Quando os itens dos pedidos são servidos, o gerente anota, ao lado do item no quadro-branco, o número de itens ou peso do prato. Quando o cliente pede a conta, o gerente se refere ao quadro-branco para calcular o valor devido. O quadro-branco é um *modelo* do restaurante, representando de forma simplificada as informações do restaurante que são necessárias para a contabilização dos pedidos feitos para os garçons e gerente.[1]

*dados em modelos*

O modelo do restaurante representa certos *dados* ou informações, que no caso são os itens e quantidade dos pedidos por mesa. Como o modelo é uma simplificação do mundo real, os dados contidos no modelo são somente os relevantes à abstração do mundo real sendo feita. Por exemplo, para a anotação dos pedidos e cálculo da conta dos clientes do restaurante, alguns dados sobre o restaurante como endereço e data da inauguração são irrelevantes, e não devem ser representadas pelo modelo em questão.

*operações em modelos*

Um modelo comumente contém *operações* ou procedimentos associados a ele. Estas operações são listas de comandos que processarão os dados contidos no próprio modelo (e em alguns casos, dados adicionais). Algumas operações que podemos fazer no modelo do Restaurante Caseiro Hipotético seriam a inclusão de um pedido para uma mesa, a modificação do *status* de um pedido de uma mesa (isto é, se o pedido foi servido ou não), o encerramento dos pedidos dos clientes de uma mesa e apresentação da conta para os clientes.

Também é possível a criação de modelos que contenham somente dados ou somente operações. Modelos que contenham somente dados são pouco usados: normalmente os valores destes dados

[1] O modelo, neste caso, é bem simples, mas representa adequadamente o tipo de artigo vendido no Restaurante Caseiro Hipotético e seu número de mesas. Modelos ou quadros-brancos similares poderiam ser criados para restaurantes bem mais sofisticados ou com mais mesas.

variam de uso para uso do modelo (ex. cada mesa do modelo do Restaurante Caseiro Hipotético pode ter quantidades de itens pedidos diferentes), portanto operações que modificam os valores dos dados serão necessárias. Quando criamos modelos para representação de dados é interessante e útil adicionar algumas operações para manipulação destes dados.

*bibliotecas de operações*

Modelos que contenham somente operações podem ser considerados *bibliotecas de operações* - exemplos são grupos de funções matemáticas e de processamento de dados que não precisem ser armazenados. A criação e uso de bibliotecas de operações em Java será vista no capítulo 5.

Para manter a convenção de nomes que será usada quando transformarmos os modelos em classes e programas em Java, de agora em diante, nomes de modelos, dados de modelos e operações serão apresentados em fonte `proporcional`, sendo que dados e operações serão com minúsculas alternadas de maiúsculas e modelos terão a primeira letra maiúscula. Considerando o exemplo mostrado, o modelo poderia ser chamado `RestauranteCaseiro`, e poderia ter os dados `quantasGramas` e `quantidadeDoItem`, e as operações `adicionaItem` e `apresentaConta`. Nomes de modelos (que serão escritos como classes) **não** devem conter acentos, que podem ser usados livremente em nomes de dados e operações.

*modelos dentro de modelos*

Modelos podem conter sub-modelos e ser parte de outros modelos: o quadro-branco que representa um restaurante pode ser composto de diversos quadrados no quadro que representam mesas ou comandas, cada uma contendo os dados relativos aos pedidos daquela mesa e ações correspondentes. O modelo `RestauranteCaseiro` poderia então conter vários exemplares do modelo `MesaDoRestaurante` que representariam diferentes mesas do restaurante. Similarmente, se um modelo `Data` for criado para representar em conjunto dados sobre um dia, mês e ano, podemos usar este modelo dentro de outros modelos que usem uma data para representar, por exemplo, um nascimento ou evento.

A simplificação inerente aos modelos é em muitos casos, necessária: dependendo do contexto, algumas informações devem ser ocultas ou ignoradas. Por exemplo, a representação das informações sobre uma pessoa pode ser feita de maneira diferente dependendo do contexto, como nos três exemplos mostrados abaixo:

- **Pessoa como Empregado de Empresa**: Para a representação uma pessoa como sendo empregado de uma empresa, para fins de processamento de folha de pagamento, seria necessária a representação do `nome`, `cargo`, `salário` e `horasExtrasTrabalhadas` desta pessoa, dentre outros dados. Este modelo poderia conter as operações `calculaSalário` e `aumentaSalário`.
- **Pessoa como Paciente de uma Clínica Médica**: Para um modelo de paciente seria necessário representar o `nome`, o `sexo`, a `idade`, a `altura`, o `peso` e o `históricoDeConsultas` do paciente. Este modelo poderia conter as operações `verificaObesidade`, que usaria os dados `sexo`, `altura` e `peso` para verificar se aquela pessoa está com o peso ideal para sua categoria e `adicionaInformaçãoAoHistórico`, que adicionaria informações recentes ao histórico do paciente.
- **Pessoa como Contato Comercial**: Para este modelo, seria necessário representar o `nome`, o `telefone`, o `cargo` e `empresa` da pessoa em questão. Algumas operações úteis para este modelo seriam `mostraTelefone` que retornaria o telefone de uma certa pessoa e `trabalhaEmEmpresa` que informaria se a pessoa trabalha em uma certa empresa.

*"super modelo"*

As três maneiras de representarmos os dados de uma pessoa e operações nestes dados (ou os três modelos de representação de pessoas) são dependentes de contexto: alguns dados e operações

que podem ser úteis para um modelo de representação de pessoas podem ser irrelevantes para outro - por exemplo, não faria sentido representar o `salário` de uma pessoa para fins de registros no banco de dados de pacientes de uma clínica ou a operação `verificaObesidade` de uma pessoa que não seja paciente de uma clínica médica. Por esta razão, e apesar dos três exemplos acima representarem pessoas de forma genérica, é difícil, se não impossível, elaborar um *"super modelo"* capaz de representar **todos** os dados e operações relativos a uma pessoa, independente do contexto, e usar este modelo no lugar de todos os outros mais especializados ou dependentes de contexto.

Modelos podem ser reutilizados para representar diferentes objetos, pessoas ou itens: o mesmo modelo `PacienteDeClinica` poderia ser utilizado para representar cada um dos pacientes de uma clínica - os pacientes podem ser representados pelo mesmo modelo, mas os dados individuais de cada um podem ser diferentes, criando a necessidade de diversos exemplos de cada modelo - assim `João`, `Pedro` e `Maria` seriam exemplos do modelo `Paciente`, cada um contendo dados de um paciente, provavelmente diferentes entre si. Vemos que não é necessária a criação de um modelo para cada item, pessoa ou objeto do mundo real.

A criação e uso de modelos é uma tarefa natural e a extensão desta abordagem à programação deu origem ao paradigma *Programação Orientada a Objetos*.

## 1.3 O que é Programação Orientada a Objetos ?

*O que é POO*

*Programação Orientada a Objetos* ou, abreviadamente, *POO*, é um paradigma de programação de computadores onde se usam classes e objetos, criados a partir dos modelos descritos anteriormente, para representar e processar dados usando programas de computadores.

Em programação orientada a objetos, os dados pertencentes aos modelos são representados por tipos de dados nativos, ou seja, que são característicos da linguagem de programação. Dados também podem ser representados por modelos já existentes na linguagem ou por outros modelos criados pelo programador. As principais diferenças entre os modelos descritos na seção 1.2 e as classes e objetos criados em programas orientados a objetos são devidas à limitação nas tarefas que um computador pode executar e às limitações da própria linguagem de programação.

Um exemplo destas diferenças pode ser analisado considerando o modelo `PacienteDeClinica`, descrito anteriormente, e a sua operação `adicionaInformaçãoAoHistórico`. Se este modelo fosse usado manualmente, ou seja, sem o uso de computadores, esta operação se resumiria a escrever em uma ficha própria as informações adicionais, que provavelmente estarão juntas às informações prévias. A implementação deste modelo e operação em programas de computador provavelmente terá que ser feita usando arquivos para armazenar as informações, e provavelmente existirão restrições quanto ao tipo e tamanho das informações a serem adicionadas - possivelmente somente informações textuais poderão ser armazenadas, mas não gráficos, ao menos nas aplicações mais simples.

Mesmo com a limitação inerente aos modelos que devem ser implementados em um computador, as vantagens de se usar computadores compensam as limitações. Ainda considerando o modelo `PacienteDeClinica`, se o dado `históricoDeConsultas` for estruturado adequadamente, seria simples criar uma operação `listaSintomas` que listaria todos os sintomas que aquele paciente já teve. Similarmente, seria simples escrever um programa que, considerando todos os pacientes da clínica, listasse os que tem potencial para hipertensão verificando, para cada paciente, os seus dados `idade`, `peso` e `históricoDeConsultas` - esta tarefa seria complicada e demorada se os

modelos fossem fichas em papel.

A modelagem dos dados e operações nestes dados em um programa de computador permite o processamento de dados de forma coesa, rápida e menos suscetível a erros. Por outro lado, mais atenção tem que ser dada ao *design* dos modelos que serão implementados, em especial às regras e limitações dos computadores e linguagens de programação. Os detalhes de implementação de modelos em Java serão mostrados no capítulo 2.

## 1.4 Encapsulamento

Vimos que modelos podem conter dados para representação das informações ou dados relativos ao que se deseja modelar e operações para manipulação destes dados. Em muitos casos será desejável que os dados não possam ser acessados ou usados diretamente, mas somente através das operações cuja especialidade será a manipulação destes dados.

Como analogia, vamos considerar uma câmera fotográfica automática. Quando um usuário da câmera clica o botão para tirar uma foto, diversos mecanismos entram em ação que fazem com que a velocidade e abertura apropriada do obturador sejam selecionadas, levando em conta o tipo do filme e as condições de iluminação. Para o usuário, os detalhes de como os dados como velocidade, abertura, tipo de filme e iluminação são processados são irrelevantes, o que interessa a ele é que a foto seja tirada.

*ocultação de dados*

O mecanismo de seleção da câmera *oculta* os dados (iluminação, tipo de filme, abertura, velocidade, etc.) e a maneira com que estes são processados, não deixando que o usuário modifique-os à vontade. A câmera deixa para o usuário somente uma maneira simplificada de efetuar os cálculos (mais exatamente, pedir à câmera que calcule os dados) e tirar a foto de acordo com os resultados obtidos: o botão que aciona o obturador.

Em muitos modelos teremos vantagens em usar um mecanismo de ocultação de dados: sempre que existir uma maneira de deixar ao modelo a capacidade e responsabilidade pela modificação de um de seus dados, devemos criar uma operação para fazê-lo. Como um exemplo, consideremos a classe `PacienteDeClínica`. Para quem estiver usando este modelo, será mais prático deixar que a operação `adicionaInformaçãoAoHistórico` esconda os detalhes de como uma informação será adicionada ao histórico de um paciente. Similarmente, para um modelo que represente uma conta bancária, a operação `retira` poderia ser encarregada de verificar se o saldo permite a retirada de um valor especificado, sem permitir acesso direto ao valor do saldo.

*encapsulamento*

A capacidade de ocultar dados dentro de modelos, permitindo que somente operações especializadas ou dedicadas manipulem estes dados ocultos chama-se *encapsulamento*, e é um dos benefícios mais palpáveis de programação orientada a objetos. Modelos que encapsulam os dados possibilitam a criação de programas com menos erros e mais clareza. Encapsulamento de dados em modelos deve ser um dos principais objetivos do programador que use linguagens orientadas a objetos.

## 1.5 Mais exemplos de modelos

Nesta seção veremos mais alguns exemplos de modelos para fixação dos conceitos. Os modelos serão descritos usando pseudo-código em português. O pseudo-código é somente uma ferramenta prática para expressão dos modelos de forma concisa e formal mas sem entrar nos detalhes de



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

implementação de classes em Java[2].

### 1.5.1 Exemplo 1: Uma lâmpada incandescente

Consideremos uma lâmpada incandescente comum e um modelo para representá-la. Esta lâmpada tem um dado básico, que é seu estado ("ligada" ou "desligada"). As operações que podemos efetuar nesta lâmpada também são simples: podemos ligá-la ou desligá-la. O ato de ligar a lâmpada equivale a modificar seu estado para "ligada", enquanto que desligar a lâmpada significa modificar seu estado para "desligada". Para saber se uma lâmpada está ligada ou desligada podemos pedir que uma operação mostre o valor do estado. Nota-se neste exemplo que a modelagem de entidades do mundo real pode ser muito diferente da que vai ser usada em programas de computador: no mundo real, para saber se a lâmpada está ligada ou não basta observá-la, enquanto que na modelagem para uso em um computador precisamos de uma operação para saber se a lâmpada está ou não ligada.

*UML*

A figura 1.2 mostra o modelo `Lampada` usando uma variante do diagrama de classes da Linguagem Unificada de Modelagem (*Unified Modeling Language*, UML). Neste diagrama, o retângulo superior mostra o nome do modelo ou classe, o retângulo central mostra que dados do modelo ou campos da classe podem ser manipulados e o retângulo inferior mostra que operações do modelo ou métodos da classe podem ser usados. Os nomes das operações ou métodos são seguidos de parênteses, e quando argumentos ou parâmetros devem ser especificados para as operações, eles são listados dentro dos parênteses.[3] Para facilitar a transição entre estes diagramas de modelos, as listagens de modelos e as classes que serão desenvolvidas a partir destes, a convenção de nomes de modelos, dados e operações será usada.

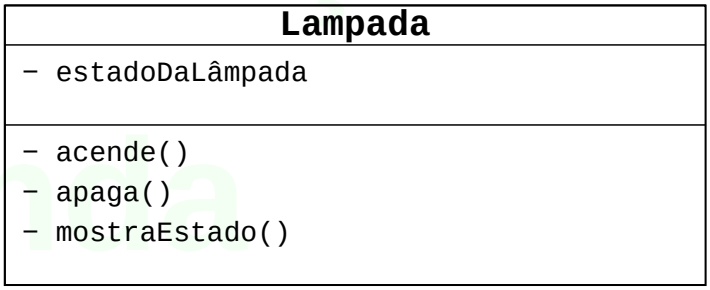



Figura 1.2: O modelo `Lampada`, seus dados e atributos

É importante ressaltar que os nomes dos dados e operações devem descrever a sua função, mesmo que com isto nomes longos apareçam nos diagramas e listagens, como no exemplo mostrado na figura 1.2.

Na figura 1.2 podemos ver que alguns dados de lâmpadas como consumo em watts, cor e tamanho foram deixados de lado. A decisão de que dados e operações devem pertencer a um modelo dependem da abrangência e escopo deste modelo: se as lâmpadas a serem representadas por este modelo fossem itens à venda em um supermercado, certamente os dados `estado` e operações `acende`, `apaga` e `mostraEstado` não seriam usados. Se o modelo da lâmpada fosse usado em uma aplicação de controle de qualidade, dados como `númeroDoLote`, `dataDeFabricação` e `unidadeProdutora` poderiam ser úteis.

[2] Considerei que seria interessante aprender antes como os modelos e classes podem ser estruturados e introduzir alguns conceitos simples do que entrar logo em detalhes de implementação e sintaxe de programas e classes em Java.

[3] Para simplificar, nestes diagramas os tipos dos argumentos para as operações nos modelos não são mostrados.

A listagem 1.1 mostra como o modelo `Lampada` poderia ser descrito em pseudo-código.

Listagem 1.1: O modelo `Lampada`, em pseudo-código.

```
1  modelo Lampada // representa uma lâmpada em uso
2  início do modelo
3    dado estadoDaLâmpada; // indica se está ligada ou não
4
5    operação acende() // acende a lâmpada
6      início
7        estadoDaLâmpada = aceso;
8      fim
9
10   operação apaga() // apaga a lâmpada
11     início
12       estadoDaLâmpada = apagado;
13     fim
14
15   operação mostraEstado() // mostra o estado da lâmpada
16     início
17       se (estadoDaLâmpada == aceso)
18         imprime "A lâmpada está acesa";
19       senão
20         imprime "A lâmpada está apagada";
21     fim
22
23 fim do modelo
```

Algumas notações e estilos que foram usadas na listagem 1.1 são:

- Existe uma distribuição vertical das linhas do texto que descreve o modelo que fazem com que a leitura seja facilitada, mesmo que a listagem acabe ocupando mais espaço. Apesar de ser permitido escrever programas de maneira mais compacta, o estilo mostrado foi adotado no resto do livro para tornar a leitura mais clara.
- Existe uma indentação horizontal das linhas do modelo que reflete a hierarquia dos componentes do modelo: dentro do modelo (contido entre as linhas 2 e 23) existem as operações e dentro das operações existem comandos. Não existem comandos ou declarações de dados fora dos delimitadores do modelo. Este estilo corresponde aproximadamente às regras de criação de classes em Java.
- Em algumas linhas, existem *comentários* que auxiliam a compreensão do trecho de código ao qual estão associados. Em um programa ou classe a ser compilada pelo compilador Java, estes comentários também servirão para facilitar a leitura do código, mas serão ignorados pelo compilador. Os comentários mostrados nestas listagens são todas as palavras que aparecem depois dos símbolos `//` (inclusive) até o final das linhas onde aparecem. *comentários*
- Cada declaração de dados e cada comando da linguagem termina com um ponto-e-vírgula (`;`). Declaração de blocos e nomes dos modelos não são terminados por ponto-e-vírgula.
- A comparação de valores é feita com dois sinais de igual (`==`), enquanto que a atribuição de valores é feita com um só sinal de igual (`=`). Na linha 7, estamos atribuindo o valor `aceso` ao dado `estadoDaLâmpada`, e na linha 17 estamos verificando se o valor `aceso` e o representado no dado `estadoDaLâmpada` são exatamente iguais.
- Blocos de código estão contidos entre as palavras `início` e `fim`, no exemplo, delimitando o modelo e as operações dentro deste modelo. O uso de blocos de código é extensivo em programação, sendo que um bloco representa uma lista de passos a ser executado sob certas condições. Por exemplo, se a operação `apaga` for chamada, o bloco entre as linhas 11 e 13 será executado. *blocos*

- Todos os dados e operações do modelo estão situados dentro dos delimitadores (`início` e `fim`) do próprio modelo.
- Como os dados são declarados dentro do modelo mas fora das operações, estes dados podem ser vistos (usados) de dentro de qualquer operação do modelo. Em outras palavras, a operação `mostraEstado` tem pleno acesso ao valor do dado `estadoDaLâmpada`.

*execução condicional*

- Alguns trechos das operações podem ser executados condicionalmente, como por exemplo trechos da operação `mostraEstado`, onde somente um dos dois comandos de impressão será executado.

Programadores acostumados com linguagens procedurais de programação como C ou Pascal devem estar curiosos para saber como este modelo poderá ser executado como um programa. Devemos lembrar que um modelo **não** é um programa, sendo apenas um mecanismo de representação de dados capaz de manipular os dados que representa através de operações. Programas (aplicações) em Java, que poderão usar as classes criadas usando estes modelos como bases, serão vistos no capítulo 3.

### 1.5.2 Exemplo 2: Uma conta bancária simplificada

O modelo deste exemplo representa uma conta bancária simplificada, que somente representa o nome do correntista, o saldo da conta e se a conta é especial ou não. Se a conta for especial, o correntista terá o direito de retirar mais dinheiro do que tem no saldo (ficar com o saldo negativo). Aspectos práticos encontrados em contas de banco reais (senhas, taxas, impostos, etc.) foram deixados de lado em nome da simplicidade. A figura 1.3 mostra os dados e operações deste modelo, e a listagem 1.2 mostra o modelo descrito em pseudo-código.

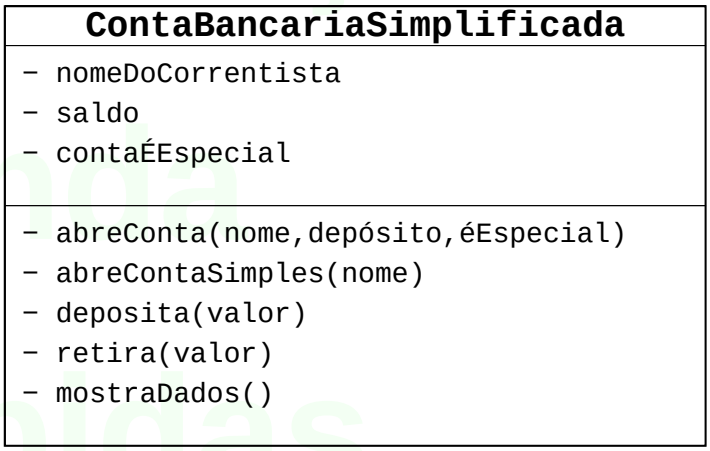



Figura 1.3: O modelo `ContaBancariaSimplificada`, seus dados e atributos

Listagem 1.2: O modelo `ContaBancariaSimplificada`, em pseudo-código.

```
1 modelo ContaBancariaSimplificada
2 início do modelo
3   dado nomeDoCorrentista,saldo,contaÉEspecial; // dados da conta
4
5   // Inicializa simultaneamente todos os dados do modelo
6   operação abreConta(nome,depósito,especial) // argumentos para esta operação
7     início
8       // Usa os argumentos passados para inicializar os dados do modelo
9       nomeDoCorrentista = nome;
```

```
10        saldo = depósito;
11        contaÉEspecial = especial;
12      fim
13
14    // Inicializa simultaneamente todos os dados do modelo, usando o nome
15    // passado como argumento e os outros valores com valores default
16    operação abreContaSimples(nome) // argumento para esta operação
17      início
18        nomeDoCorrentista = nome;
19        saldo = 0.00;
20        contaÉEspecial = falso;
21      fim
22
23    // Deposita um valor na conta
24    operação deposita(valor)
25      início
26        saldo = saldo + valor;
27      fim
28
29    // Retira um valor da conta
30    operação retira(valor)
31      início
32        se (contaÉEspecial == falso) // A conta não é especial !
33          início
34            se (valor <= saldo) // se existe saldo suficiente...
35              saldo = saldo - valor; // faz a retirada.
36          fim
37        senão // A conta é especial, pode retirar à vontade !
38          saldo = saldo - valor;
39      fim
40
41    operação mostraDados() // mostra os dados da conta, imprimindo os seus valores
42      início
43        imprime "O nome do correntista é ";
44        imprime nomeDoCorrentista;
45        imprime "O saldo é ";
46        imprime saldo;
47        se (contaÉEspecial) imprime "A conta é especial.";
48        senão imprime "A conta é comum.";
49      fim
50
51  fim do modelo
```

Alguns pontos interessantes da listagem 1.2 são:

- Existem duas operações que podem ser usadas para a abertura de contas: uma para a qual temos que passar todos os dados que serão usados (`abreConta`) e outra para a qual somente precisaremos passar o nome do correntista (`abreContaSimples`). É esperado que somente uma destas operações seja usada cada vez que o modelo for usado, mas oferecer várias opções de uso para quem for utilizar o modelo é comum.
- O conceito de que operações manipulam os dados do modelo fica mais claro ainda com as operações `deposita` e `retira`, que são usadas para modificar o saldo da conta.
- O uso de blocos é demonstrado de novo na operação `retira`, onde um bloco delimita comandos que só serão executados se a conta não for especial.

### 1.5.3 Exemplo 3: Uma data

Outro modelo simples que pode ser criado facilmente é o modelo `Data`, para representação de datas de eventos como aniversários, eventos, etc., que deve conter dados para representar o dia,



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

o mês e o ano. A figura 1.4 mostra os dados e operações neste modelo.

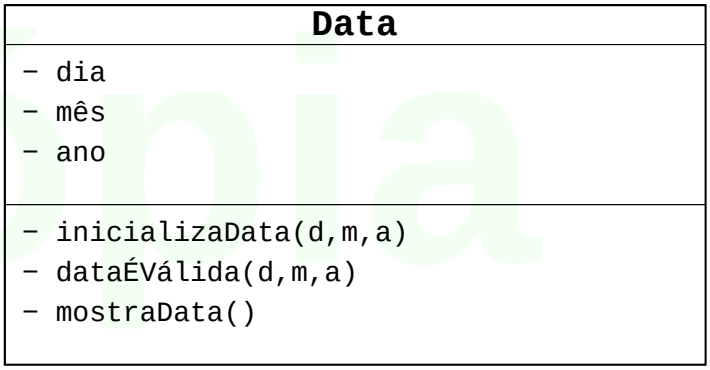



Figura 1.4: O modelo `Data`, seus dados e atributos

Para este modelo, supomos que o valor do mês é representado por um número inteiro entre 1 e 12, inclusive (ao invés de ser representador pelo nome em extenso do mês, por exemplo). Consideraremos também que existem datas *válidas* e *não-válidas*, pois os dados que representam o dia, mês e ano podem ter combinações válidas (ex. 25,10,2000 ou 19,07,1970) e não-válidas (ex. 31,02,2000 ou -5,-2,1999). O modelo deve ter uma operação que identifique datas não-válidas e não as represente como se fossem válidas, ou seja, uma maneira simples de diferenciar datas válidas das não-válidas. A maneira escolhida de fazer isto foi dizer que se o dia e/ou o mês forem iguais a zero, a data não será válida. O modelo `Data`, que representa os dados e as operações da figura 1.4, é mostrado na listagem 1.3.

Listagem 1.3: O modelo `Data`, em pseudo-código.

```
1  modelo Data
2  início do modelo
3    dado dia,mês,ano; // componentes da data
4
5    // Inicializa simultaneamente todos os dados do modelo
6    operação inicializaData(umDia,umMês,umAno) // argumentos para esta operação
7      início
8        // Somente muda os valores do dia, mês e ano se a data passada for válida
9        se dataÉVálida(umDia,umMês,umAno) // Repassa os argumentos para a operação
10         início
11           dia = umDia;
12           mês = umMês;
13           ano = umAno;
14         fim
15       // Se a data passada não for válida, considera os valores sendo zero
16       senão
17         início
18           dia = 0;
19           mês = 0;
20           ano = 0;
21         fim
22     fim
23
24   operação dataÉVálida(umDia,umMês,umAno) // argumentos para esta operação
25     início
26       // Se a data passada for válida, retorna verdadeiro
27       se ((dia >= 1) e (dia <= 31) e (mês >= 1) e (mês <= 12))
28         retorna verdadeiro;
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
29        // Senão, retorna falso
30        senão
31          retorna falso;
32      fim
33
34    operação mostraData() // mostra a data imprimindo valores de seus dados
35      início
36        imprime dia;
37        imprime "/";
38        imprime mês;
39        imprime "/";
40        imprime ano;
41      fim
42
43  fim do modelo
```

Alguns pontos interessantes do modelo mostrado na listagem 1.3 são:

- Uma vantagem do encapsulamento que podemos ver neste modelo é a restrição aos valores que podem ser representados pelos dados: datas não-válidas são identificadas e os valores dos dados `dia`, `mês` e `ano` são inicializados correspondentemente, de forma que se a data passada não for válida, os dados sempre valerão zero.[4]
- Uma vez que os dados a serem representados pelo modelo `Data` não podem assumir qualquer valor, é desejável que estes dados sejam manipuláveis somente pelas operações do modelo. Desta forma, ou os dados encapsulados contém uma data válida ou todos contém zero - não é possível ter o dia e o ano corretos e o mês incorreto.
- A operação `inicializaData` recebe argumentos, e usa os valores passados como argumentos para efetuar alguma operação, neste caso, manipulação dos dados do modelo. Se não fosse possível passar argumentos para a operação, não teríamos como inicializar os dados do modelo.
- Podemos ver que também é possível chamar operações de dentro de operações como na linha 9 da listagem 1.3. Quando a operação `dataÉVálida` é chamada, o seu resultado é verificado e parte da operação `inicializaData` é executada dependendo do resultado. Os argumentos passados para a operação `dataÉVálida` são os mesmos passados para a operação `inicializaData` - a operação `inicializaData` *delegou* a verificação de validade à operação `dataÉVálida`. Desta forma, quando o modelo for usado, poderemos inicializar uma data diretamente ou verificar se ela é válida sem precisar inicializar.
- A operação `dataÉVálida` retorna um valor à operação que a chamou através da palavra-chave `retorna` do pseudo-código.
- Podemos imprimir mensagens fixas com o comando de pseudo-código `imprime` se colocarmos a mensagem entre aspas, e imprimir valores dos dados usando o nome dos dados.

### 1.5.4 Exemplo 4: Um registro acadêmico de aluno

Consideremos agora como representar os dados acadêmicos básicos de um aluno de uma escola em um modelo. Este modelo pode ser considerado como uma ficha de matrícula ou identificação do aluno na escola, e como os modelos apresentados até agora, é apenas uma versão simplificada - modelos mais complexos, contendo mais informações, seriam mais próximos das fichas reais de matrícula de escolas. Os dados e operações do modelo proposto são mostrados na figura 1.5.

[4] Evidentemente este modelo é uma simplificação incorreta de datas, pois permite que sejam criadas datas como 31/02/2000. No momento, consideremos este modelo como sendo uma versão simplificada e incorreta, alguns exercícios propostos no fim deste capítulo sugerem as modificações para que o modelo fique correto.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

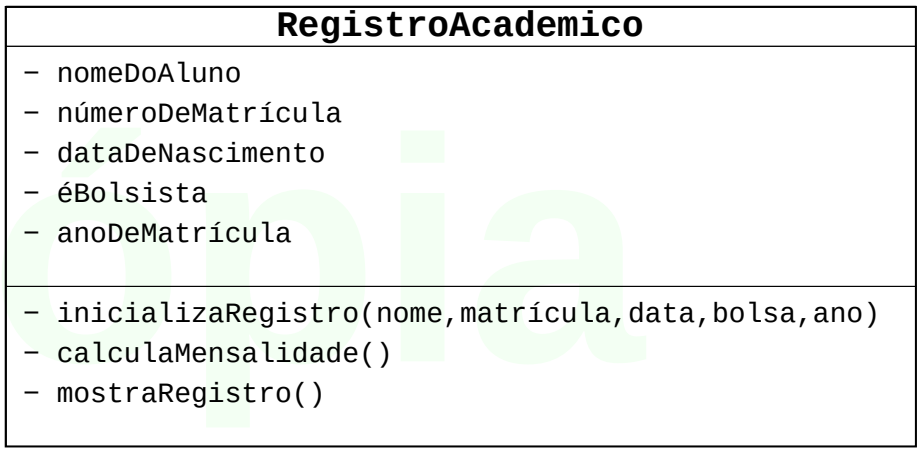



Figura 1.5: O modelo `RegistroAcademico`, seus dados e atributos

Um dos dados do modelo representa uma data de nascimento. Ao invés de usar dados para representar separadamente o dia, mês e ano desta data, podemos aproveitar o modelo `Data`, visto na seção anterior, para representar a data de nascimento. Desta forma, ao invés de reinventarmos a roda, poderemos aproveitar as características do modelo `Data` de forma simples (somente um único dado será manipulado ao invés de três) e segura (mecanismos de identificação de datas válidas e não-válidas já estão implementadas no modelo `Data`). A listagem 1.4 mostra a descrição dos dados e operações propostas para o modelo `RegistroAcademico`.

Listagem 1.4: O modelo `RegistroAcademico`, em pseudo-código.

```
1  modelo RegistroAcademico
2  início do modelo
3    // Dados do registro acadêmico
4    dado nomeDoAluno,númeroDeMatrícula;
5    dado dataDeNascimento,éBolsista,anoDeMatrícula;
6
7    // Inicializa simultaneamente todos os dados do modelo, passando argumentos
8    operação inicializaRegistro(oNome,aMatrícula,aData,temBolsa,qualAno)
9      início
10       // Usa os argumentos para inicializar os valores no modelo
11       nomeDoAluno = oNome;
12       númeroDeMatrícula = aMatrícula;
13       dataDeNascimento = aData;
14       éBolsista = temBolsa;
15       anoDeMatrícula = qualAno;
16     fim
17
18   operação calculaMensalidade() // calcula e retorna a mensalidade
19      início
20        mensalidade = 400;
21        se (éBolsista) mensalidade = mensalidade / 2;
22        retorna mensalidade;
23      fim
24
25   operação mostraRegistro() // mostra os dados do registro acadêmico
26     início
27       imprime "Nome do aluno:";
28       imprime nomeDoAluno;
29       imprime "Número de Matrícula:";
30       imprime númeroDeMatrícula;
31       imprime "Data de Nascimento:";
32       dataDeNascimento.mostraData(); // pede à data que se imprima !
```

```
33      se (éBolsista == verdadeiro) imprime "O aluno é bolsista.";
34      senão imprime "O aluno não é bolsista.";
35      imprime "Ano de Matrícula:";
36      imprime anoDeMatrícula;
37    fim
38
39 fim do modelo
```

Alguns pontos interessantes da listagem 1.4 são:

- Temos agora uma operação (`calculaMensalidade`) que retorna um valor que é calculado dentro da operação, usando os dados encapsulados no modelo e um algoritmo simples. O uso de uma operação para calcular um valor é outra vantagem do encapsulamento: deixa-se a cargo do modelo o cálculo de valores associados a ele.
- Na operação `mostraRegistro`, ao invés de usar o comando em pseudo-código `imprime` para imprimir o valor da data de nascimento, pedimos à data que se imprima. Como o dado `dataDeNascimento` é representado pelo modelo `Data`, podemos usar a sua operação `mostraData` para pedir a impressão dos dados da data. Por convenção, o conteúdo de um modelo (dados e operações) pode ser acessado usando um ponto (`.`) entre o nome do exemplo do modelo e o dado ou operação encapsulada por este modelo. Também por convenção, quando uma operação de um modelo for chamada, parênteses são usados para especificar que argumentos serão passados. Como nenhum argumento é passado para a operação `mostraData`, parênteses vazios (sem conteúdo) são usados.

*operador ponto (.)*

## 1.6 Classes, objetos, instâncias e referências

Programadores que utilizam o paradigma de Programação Orientada a Objetos criam e usam *objetos* a partir de *classes*, que são relacionadas diretamente com os modelos descritos anteriormente.

*Classes* são estruturas das linguagens de programação orientadas a objetos para conter, para determinado modelo, os dados que devem ser representados e as operações que devem ser efetuadas com estes dados. Cada classe deve ter um nome que seja facilmente associável ao modelo que a classe representa.

*classes*

Classes são escritas com os recursos e regras da linguagem de programação orientada a objetos para implementação dos modelos, mas em muitos dos casos as classes são somente moldes ou formas que representam os modelos abstratamente. Para representação de dados específicos usando classes deste tipo será necessária a criação de *objetos* ou *instâncias* desta classe. Um objeto ou instância é uma materialização da classe, e assim pode ser usado para representar dados e executar operações. Para que os objetos ou instâncias possam ser manipulados, é necessária a criação de *referências* a estes objetos, que são basicamente variáveis do "tipo" da classe.[5]

*objetos*
*instâncias*
*referências*

Fazendo uma analogia, uma classe poderia ser considerada como sendo uma planta de prédio, que descreve o prédio mas não corresponde fisicamente a ele, enquanto que os prédios construídos de acordo com aquela planta seriam as instâncias. Similarmente, uma ficha de matrícula em branco, sem conter informações, seria uma classe enquanto que várias fichas daquele tipo, preenchidas, corresponderiam à instâncias daquela classe. O nome ou número de cada um destes

[5] Os objetos são armazenados na pilha (*heap*) do sistema, e as referências apontam para os objetos. Para programadores em C ou C++, referências podem ser consideradas como ponteiros, mas referências em Java não tem a complexidade de alocação e manipulação existente em outras linguagens.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

prédios seria a referência às instâncias da planta do prédio.

Considerando os exemplos de modelos mostrados anteriormente neste capítulo, os modelos seriam classes e a partir destas classes poderíamos criar instâncias. Por exemplo, se o modelo `Lampada` fosse escrito como uma classe, poderíamos ter as referências `lâmpadaDaSala`, `lâmpadaDoQuarto` e `lâmpadaDaCozinha` às instâncias da classe `Lampada`. Da mesma forma, poderíamos ter as referências `hoje`, `diaDaIndependência` e `dataDeNascimento` às instâncias da classe `Data`.

*campos*
*tipo de dado*
*variáveis*

Os dados contidos em uma classe são conhecidos como *campos* ou *atributos* daquela classe. Cada campo deve ter um nome e ser de um *tipo*, que será ou um tipo de dado nativo da linguagem Java ou uma classe existente na linguagem ou definida pelo programador. Valores contidos dentro de classes também são chamados de *variáveis*, sendo que campos representam dados encapsulados em uma classe, e variáveis representam valores auxiliares necessários ao funcionamento dos métodos na classe.

Se a classe é usada para que várias instâncias sejam criadas a partir dela, cada uma destas instâncias terá um conjunto dos campos definidos na classe. Se, por exemplo, a classe `Aluno` for criada, e conter os campos `nome`, `matrícula` e `anoLetivo`, cada uma das instâncias da classe `Aluno` terá os seus campos `nome`, `matrícula` e `anoLetivo`, que poderão receber valores diferentes[6].

*métodos*

As operações contidas em uma classe são chamadas de *métodos* desta classe. Métodos são geralmente chamados ou executados explicitamente a partir de outros trechos de código (comumente, outros métodos) na classe que o contém ou a partir de outras classes. Por exemplo, se o modelo `RegistroAcademico` possui uma referência a uma instância da classe `Data` que representa a data de nascimento do aluno, quando o método `imprimeDados` da classe `RegistroAcademico` for chamado, este pode por sua vez chamar o método `imprimeData`, declarado na classe `Data`, para imprimir os dados da data de nascimento do aluno (veja a listagem 1.4). Similarmente, um programa pode pedir aos métodos de um modelo que se executem para obter um determinado resultado. Muitos autores preferem usar o termo *mensagens* para se referir à chamada de métodos: ao executar um método estaríamos enviando uma mensagem a este método.

*argumentos*

Métodos podem opcionalmente receber *argumentos para métodos* na forma de valores de tipos nativos de dados ou referências à instâncias de classes. Vários argumentos de vários tipos podem ser fornecidos simultaneamente para um método. Os argumentos devem ser declarados quando declaramos o método.

*retorno de valores por métodos*

Métodos podem também opcionalmente retornar valores ou instâncias de classes. Métodos podem não retornar nenhum valor ou retornar um único valor, mas não podem retornar simultaneamente mais do que um valor.

O processamento de dados que deve ser feito para um modelo será feito dentro dos métodos, que eventualmente poderão executar (ou chamar) outros métodos, da mesma classe a que pertencem ou de outras classes.

[6] Esta explicação sobre campos em instâncias diferentes da mesma classe poderem receber valores diferentes é uma simplificação necessária, mais detalhes e exceções a esta regras serão vistos no capítulo 5.

## 1.7 Orientação a Objetos é necessária ?

Às vezes o modelo da tarefa a ser executada é tão simples que a criação de classes baseadas no modelo torna a solução mais complicada, especialmente quando não existe necessidade ou vantagem na criação de várias instâncias da mesma classe. Um exemplo simples é a modelagem da resolução de uma equação de segundo grau: o problema pode ser facilmente declarado como um algoritmo simples e implementado em uma linguagem procedural, que resolva o algoritmo passo a passo, sem a necessidade de orientação a objetos. Entretanto, o problema também pode ser resolvido, ainda que de forma um pouco mais complexa, usando orientação a objetos.

Basicamente, qualquer aplicação onde seja necessário representar conjuntos de dados dependentes ou interligados entre si (ex. datas, registros de alunos, contas bancárias, etc.) podem se beneficiar das facilidades de encapsulamento das linguagens de programação orientadas a objetos. Similarmente, qualquer aplicação onde seja necessário aplicar um conjunto de rotinas específicas a um conjunto de dados pode ser modelada com sucesso usando orientação a objetos.

## 1.8 Papéis de programadores que utilizam orientação a objetos

*programadores de classes*

No desenvolvimento de programas e classes usando programação orientada a objetos, encontraremos dois papéis para os programadores envolvidos. Um é o de *programadores de classes*, que são os programadores que deverão criar novas classes, considerando que campos e métodos devem ser encapsulados nesta classe para que ela represente adequadamente o modelo sendo considerado.

Os programadores de classes deverão ter cuidados especiais com que métodos permitirão a manipulação dos campos da classe - estes métodos são o *contrato* entre criador e usuários da classe. O programador deve tentar prever as aplicações que a classe poderá ter para diversos usuários, dando à classe a funcionalidade esperada mas sem sobrecarregá-la, em especial, evitando a criação de classes que representem super-modelos.

*programadores-usuários*

A segunda categoria de programadores envolvidos com programação orientada a objetos é a de *programadores-usuários* - estes programadores utilizarão classes em Java que foram feitas por outros programadores ou que são parte do conjunto de classes pré-existente na linguagem para criar aplicações.

Em muitos casos não haverá uma distinção precisa entre os dois papéis - será comum que um programador de classes crie aplicativos que demonstrem o uso das classes criadas, para testes, documentação ou aplicação final, e também será comum que programadores-usuários criem classes para representar modelos específicos a sua aplicação e que não tenham sido criadas anteriormente, ou até que modifique ou adapte classes existentes a sua aplicação.

É importante para o estudante de uma linguagem de programação orientada a objetos que se coloque nos dois papéis simultaneamente, desta forma ele ou ela deverá ser capaz de criar classes que possam ser usadas com praticidade.

## 1.9 Exercícios do capítulo 1

Os exercícios deste capítulo são puramente teóricos, e sua solução não requer conhecimentos da sintaxe da linguagem Java. Para melhor resultado as soluções encontradas devem ser discutidas com outros estudantes.

Para os exercícios relacionados com criação de modelos, é sugerida a criação também dos diagramas mostrados nas seções 1.5.1 a 1.5.4.

**Exercício 1.1:** ⋆

Descreva, com suas próprias palavras, a operação `calculaConta` do modelo que representa o Restaurante Caseiro Hipotético.

**Exercício 1.2:** ⋆

Imagine que o Restaurante Caseiro Hipotético facilite aos seus clientes a divisão dos valores da conta pelo número de clientes. Que dados adicionais deveriam ser representados pelo modelo ? Quais operações deveriam ser criadas e/ou modificadas ?

**Exercício 1.3:** ⋆

Explique, com exemplos, por que seria complicado usar um "super modelo" que representaria **todos** os dados de uma pessoa.

**Exercício 1.4:** ⋆

Escreva um modelo para representar uma lâmpada que está à venda em um supermercado. Que dados devem ser representados por este modelo ?

**Exercício 1.5:** ⋆

Imagine uma lâmpada que possa ter três estados: apagada, acesa e meia-luz. Usando o modelo `Lampada` (listagem 1.1) como base, escreva o modelo `LampadaTresEstados`.

**Exercício 1.6:** ⋆

Generalize o modelo `LampadaTresEstados` (exercício 1.5) para que ele possa representar uma lâmpada onde a luminosidade pode ser ajustada com qualquer valor entre 0% (apagada) e 100% (acesa). *Dica:* ao invés de operações para possibilitar o ajuste para cada um dos estados, descreva uma operação que receba um valor de ajuste.

**Exercício 1.7:** ⋆

Inclua, no modelo `Lampada` (listagem 1.1), uma operação `estáLigada` que retorne verdadeiro se a lâmpada estiver ligada e falso caso contrário.

**Exercício 1.8:** ⋆

A operação `abreConta` do modelo `ContaBancariaSimplificada` (listagem 1.2) permite que alguém crie uma conta bancária passando como argumento um valor negativo, criando uma conta já em débito. Modifique a operação `abreConta` para que se alguém passar um saldo inicial negativo, que este seja considerado como zero.

**Exercício 1.9:** ⋆

Modifique a operação `mostraDados` do modelo `ContaBancariaSimplificada` (listagem 1.2) para que, caso o saldo esteja negativo, uma mensagem de alerta seja impressa. *Dica:* O saldo só poderá ser negativo se a conta for especial.

**Exercício 1.10:** ⋆

Baseado no modelo `Data` (listagem 1.3) crie o modelo `HoraAproximada`, que represente uma hora qualquer (usando valores para representar horas e minutos). Que dados e operações este modelo deve ter ?

**Exercício 1.11:** ⋆

Baseado no modelo `Data` (listagem 1.3) e considerando o exercício 1.10, crie o modelo `HoraPrecisa`, que represente uma hora qualquer (usando valores para representar horas, minutos, segundos e centésimos de segundos). Que dados e operações este modelo deve ter ? Que dados e operações poderiam ser copiados do modelo `HoraAproximada`, do exercício 1.10 ?

**Exercício 1.12:** ⋆

Crie um modelo `DataHora` que represente simultaneamente uma data e uma hora aproximada. *Dica:* o modelo pode conter instâncias dos modelo `Data` e `HoraAproximada`.

**Exercício 1.13:** ⋆

O modelo `Data` (listagem 1.3) pode conter datas não válidas, com os valores de dia, mês e ano iguais a zero, que podem ser criadas quando a operação `inicializaData` for chamado com valores incorretos. Modifique a operação `mostraData` para que se o dia, mês ou ano forem inválidos (isto é, iguais a zero) uma mensagem “Data Inválida” seja impressa ao invés dos valores de dia, mês e ano.

**Exercício 1.14:** ⋆

A operação `inicializaRegistro` do modelo `RegistroAcademico` (listagem 1.4) não verifica nenhum dos dados passados, simplesmente copiando-os para os valores do modelo. Modifique esta operação para que se o ano de matrícula for menor do que 1990, seja considerado como sendo zero (inválido).

**Exercício 1.15:** ⋆

Crie um modelo `Livro` que represente os dados básicos de um livro, sem se preocupar com a sua finalidade.

**Exercício 1.16:** ⋆

Usando o resultado do exercício 1.15 como base, crie um modelo `LivroDeLivraria` que represente os dados básicos de um livro que está à venda em uma livraria. Veja também o exercício 1.17.

**Exercício 1.17:** ⋆

Usando o resultado do exercício 1.15 como base, crie um modelo `LivroDeBiblioteca` que represente os dados básicos de um livro de uma biblioteca, que pode ser emprestado a leitores. Veja também o exercício 1.16.

**Exercício 1.18:** ⋆

Usando o resultado do exercício 1.15 como base, crie um modelo `DicionarioBilingue` que represente os dados básicos de um dicionário de línguas (ex. português-inglês, latim-aramaico, etc).

**Exercício 1.19:** ⋆

Usando o resultado do exercício 1.15 como base, crie um modelo `LivroComSoftware` que represente os dados básicos de um livro acompanhado de software. *Dica:* o software pode ser demonstrativo ou não e o título pode ser diferente do título do livro.

**Exercício 1.20:** ⋆

Crie um modelo para representar um professor de uma disciplina qualquer. *Dica:* Use, para orientação, o modelo `RegistroAcademico` (listagem 1.4).

**Exercício 1.21:** ⋆

Crie um modelo para representar um time de um esporte qualquer em um campeonato deste esporte. Que dados e operações este modelo deve ter ?

**Exercício 1.22:** ⋆

Crie um modelo `Musica` para representar uma música, para uso em uma coleção ou banco de dados de músicas. Que dados e operações este modelo deve ter ?

**Exercício 1.23:** ⋆

Crie um modelo `Ponto2D` para representar um ponto no espaço cartesiano de duas dimensões. Que dados e operações este modelo deve ter ? *Dica:* Imagine um gráfico no qual você tenha que desenhar pontos, baseados neste modelo.

**Exercício 1.24:** ⋆

Crie um modelo `Veiculo` que represente os dados e operações de um veículo de transporte como nome, número de passageiros que pode carregar, tipo (aéreo, terrestre ou aquático), número de rodas, etc.

**Exercício 1.25:** ⋆

Baseado no exercício 1.24 crie um modelo `VeiculoTerrestre` que tenha os mesmos dados do modelo `Veiculo`. Quais serão as principais diferenças entre o modelo criado neste exercício e o modelo `Veiculo` ?

**Exercício 1.26:** ⋆

Baseado nos exercícios 1.24 e 1.25 crie um modelo `AutomovelDePasseio` que tenha os mesmos dados do modelo `VeiculoTerrestre`. Quais serão as principais diferenças entre o modelo criado neste exercício e o modelo `VeiculoTerrestre` ?

**Exercício 1.27:** ⋆

Escreva um modelo `Empregado` que represente um empregado de uma empresa qualquer. Considere que os dados `nome`, `departamento`, `horasTrabalhadasNoMês` e `salárioPorHora` devam ser representados, e que ao menos as operações `mostraDados` e `calculaSalárioMensal` sejam implementadas.

**Exercício 1.28:** ⋆⋆

Modifique a operação `mostraData` no modelo `Data` (listagem 1.3) para que o mês seja mostrado por extenso. *Dica:* Veja o exercício 1.13.

**Exercício 1.29:** ⋆⋆

Imagine que os empregados de uma empresa tenham dois valores de salário para horas trabalhadas, diferenciados entre horas normais e horas extras. Modifique o modelo `Empregado` (veja o exercício 1.27) para que dois valores de horas trabalhadas e dois valores de salário-hora sejam usados.

**Exercício 1.30:** ⋆⋆

Modifique a operação `calculaSalárioMensal` no modelo `Empregado` (veja o exercício 1.27) para que todos os empregados do departamento Diretoria tenham 10% de bônus salarial.

**Exercício 1.31:** ⋆⋆

Imagine que o Restaurante Caseiro Hipotético deva ser representado para fins de cálculo de impostos. Quais dados e ações devem ser representados pelo modelo ?

**Exercício 1.32:** ⋆⋆

Imagine que o Restaurante Caseiro Hipotético deva ser representado para fins de inclusão em guias turísticos. Quais dados e ações devem ser representados pelo modelo ?

**Exercício 1.33:** ⋆⋆

Crie um modelo para representar uma linha, unida por dois pontos no espaço cartesiano de duas dimensões, usando o modelo criado no exercício 1.23. Que dados e operações este modelo deve ter ?



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 1.34:** ⋆⋆

Crie um modelo para representar um retângulo, cujos pontos opostos sejam instâncias do modelo `Ponto2D` (exercício 1.23). Veja também o exercício 1.33.

**Exercício 1.35:** ⋆⋆

Considerando o exercício 1.21, escreva um modelo `Partida` que represente uma partida entre dois times do esporte. Como seria a operação `imprimeVencedor` deste modelo ?

**Exercício 1.36:** ⋆⋆

Escreva um modelo que represente um polígono regular de até dez lados. Que dados e operações este modelo deve conter ? Descreva, para este modelo, uma operação que retorne o nome do polígono baseado no seu número de lados.

**Exercício 1.37:** ⋆⋆⋆

Considere o modelo `Lampada` mostrado em pseudo-código na listagem 1.1. Imagine que uma lâmpada representada por este modelo possa ter um outro dado, `queimada`, além do dado `estado`. Que operações deveriam ser modificados no modelo `Lampada` ? Que outras operações deveriam ser adicionados ?

**Exercício 1.38:** ⋆⋆⋆

Suponha que uma determinada escola ofereça três tipos de bolsa para alguns de seus estudantes: 30% de desconto, 70% de desconto e bolsa integral com 100% de desconto. Que modificações deveriam ser feitas no modelo `RegistroAcademico` (listagem 1.4) ?

**Exercício 1.39:** ⋆⋆⋆

Usando o exercício 1.38 como base, considere que a escola também tenha cursos diurnos e noturnos, e ofereça para todos os estudantes, indiferente de terem ou não bolsa, desconto de 20% para os estudantes do tuno da manhã. Que modificações deveriam ser feitas no modelo `RegistroAcademico` (listagem 1.4) ? *Dica:* Um novo dado para representar o turno de estudos (diurno ou noturno) deverá ser criado e manipulado.

**Exercício 1.40:** ⋆⋆⋆

Crie um modelo `CDDeMusicas` que contenha várias instâncias do modelo `Musica` (exercício 1.22). Como você acha que podemos fazer para representar, em um `CDDeMusicas`, um número variável de instâncias de `Musica` ?

**Exercício 1.41:** ⋆⋆⋆

Crie um modelo `EquacaoSegundoGrau` que contenha somente uma operação, a que calcula as raízes da equação. Considere que os valores de $a$, $b$ e $c$ serão passados para uma operação deste modelo. Qual a complexidade adicional de se criar este modelo, quando comparado com um algoritmo simples ? Quais as vantagens esperadas ?

**Exercício 1.42:** ★★★★

A operação `inicializaData` do modelo `Data` (listagem 1.3) tem uma abordagem simplista demais para verificar se o dia sendo usado é válido ou não: nesta operação ainda seria possível passar a data 31/02/2000 e a operação iria considerar os valores passados como sendo válidos. Modifique a operação `dataÉVálida` para que esta considere o valor máximo que pode ser aceito como válido dependendo do mês, de forma que para meses com 30 dias, o valor 31 para o dia seja considerado incorreto, e que para Fevereiro o valor máximo seja calculado em função do ano ser bissexto ou não. *Dica:* Anos bissextos (tendo 29 dias em fevereiro) são divisíveis por quatro, a não ser que sejam divisíveis por 100. Anos que podem ser divididos por 400 também são bissextos. Desta forma, 1964 e 2000 são bissextos, mas 1900 não é bissexto. A operação de divisibilidade pode ser implementada pela função módulo, representada pelo sinal `%`, e comparada com zero: a expressão `(1966 % 4) == 0` é verdadeira, enquanto que a expressão `(1967 % 4) == 0` é falsa.

# Capítulo 2

# Criando Classes em Java

## 2.1 Introdução

No capítulo 1 vimos que o paradigma de orientação a objetos baseia-se na criação de modelos que representam entidades, pessoas, processos, etc. do mundo real, de forma simplificada. Para os exemplos vistos até agora, usamos um tipo de pseudo-código simples, que declara os dados e operações dos modelos descritos.

Neste capítulo veremos como modelos podem ser implementados em Java através de classes. As principais diferenças entre os modelos em pseudo-código vistos anteriormente e as classes em Java serão vistas gradualmente com os exemplos apresentados, que também mostrarão a sintaxe básica de alguns operadores e classes da linguagem.

Para melhor aproveitamento do livro neste capítulo em diante, será necessário ter acesso a um computador com o compilador e interpretador Java instalados e um editor de textos apropriado. Veja no apêndice B.1 informações sobre onde o ambiente gratuito de desenvolvimento pode ser encontrado para *download* e sugestões de softwares adicionais.

## 2.2 Sintaxe básica de criação de classes em Java

*regras básicas de sintaxe*

Como o código que será escrito pelo programador (contendo uma ou mais classes em Java) será compilado pelo compilador Java, este espera que o código siga certas regras básicas de sintaxe, apresentadas a seguir:

*regras para nomes de classes*

- Uma classe em Java é sempre declarada com a palavra-chave `class` seguida do nome da classe.O nome da classe não pode conter espaços, deve sempre ser iniciado por uma letra. Para nomes de classes, métodos e campos em Java, o caractere sublinhado (_) e o sinal `$` são considerados letras. O nome da classe não deve conter acentos[1] e pode conter números, contanto que estes apareçam depois de uma letra. Nomes de classes não podem ser exatamente iguais às palavras reservadas de Java (mostradas na tabela 2.1).
- Caracteres maiúsculos e minúsculos são **diferenciados** em Java: as palavras `Class`, `CLASS`, `ClAsS` e `class` são consideradas como sendo diferentes, e somente a última pode ser usada para declarar uma classe. A falha em escrever corretamente os identificadores esperados em Java pode resultar em erros de compilação.

[1] Teoricamente o nome da classe pode conter acentos, mas uma vez que o compilador irá criar arquivos com nomes baseados nos nomes das classes, podem haver problemas com estes nomes de arquivos, dependendo das limitações do sistema operacional. É aconselhável que os nomes de classes não sejam acentuados.

- O conteúdo das classes é delimitado pelas chaves (caracteres `{` e `}`) - todos os campos e métodos da classe devem estar entre estes caracteres. Blocos de código correspondentes a métodos também devem estar entre estes caracteres. A cada caracter `{` que abre um bloco deve haver um caracter `}` correspondente que fecha o bloco, caso contrário erros de compilação ocorrerão.

*blocos de código de métodos*

- Não existem regras especiais para o espaçamento vertical ou para a indentação[2] horizontal, porém espaçamento adequado e indentação regular fazem com que o código fique claro e fácil de ler.

*indentação*

A tabela 2.1 lista as palavras reservadas de Java (isto é, com significado especial para o compilador).

*palavras reservadas*

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

Tabela 2.1: Palavras reservadas em Java

Um exemplo de classe sintaticamente válida em Java é mostrado na listagem 2.1.

Listagem 2.1: Uma classe vazia em Java.

```
1  /**
2   * A classe Vazia, que não possui campos nem métodos, mas mesmo assim pode ser usada
3   * para exemplificar as regras sintáticas básicas de Java, podendo até mesmo ser
4   * compilada.
5   */
6  class Vazia // esta é a declaração da classe !
7    {
8    /* Se houvessem campos ou métodos para a classe Vazia, eles deveriam ser
9       declarados aqui dentro. */
10
11   } // fim da classe Vazia
```

Outras características sintáticas de Java e convenções de código, que podem ser vistas na listagem 2.1, são:

- Tradicionalmente os nomes de classes começam com caracteres maiúsculos e alternam entre palavras. Nomes de classes que seguem esta convenção são `RegistroAcademico`, `ContaDeLuz` e `CadastroDeFuncionariosDeSupermercado`.

*convenções para nomes de classes*

- Existem três tipos de comentários em Java, que serão ignorados pelo compilador mas podem ser bastante úteis para programadores com acesso ao código-fonte da classe.

*tipos de comentários*

  - Comentários de uma única linha começam com duas barras inclinadas (`//`) e terminam ao final da linha: servem para especificar comentários sobre uma região próxima do código, geralmente a mesma linha, como nas linhas 6 e 11 da listagem 2.1.

[2] Alguns autores usam os termos *identar* ou *endentar* para o ato de adicionar espaços ou tabulações na horizontal para que o código fique mais legível e estruturado visualmente.

  - Comentários em bloco são delimitados pelos conjuntos de caracteres `/*` no início e `*/` no fim - tudo entre estes dois conjuntos de caracteres será considerado como comentários pelo compilador. Um exemplo pode ser visto entre as linhas 8 e 9 da listagem 2.1.

*comentários para documentação*

  - Comentários em bloco para documentação são similares aos comentários em bloco comuns, exceto que são iniciados pela seqüência `/**` ao invés de `/*`. Estes comentários podem ser analisados pela ferramenta `javadoc` para criação automática de documentação para a classe. Esta ferramenta e algumas convenções de comentários serão apresentadas no apêndice **??**. As listagens de classes apresentadas neste livro contém informação adicional para documentação automática.

- É aconselhável que os arquivos criados em editores de texto contenham somente uma classe, e que os nomes dos arquivos sejam compostos dos nomes das classes com a extensão `.java`. Desta forma, a classe `Vazia` deve ser criada em um arquivo `Vazia.java`.

## 2.3 Campos em classes em Java

*declaração de campos*
*tipo de dado*

Campos de classes em Java devem ser declarados dentro do corpo da classe (a parte entre as chaves `{` e `}`). Quando tentamos representar modelos do mundo real usando linguagens de programação, cada campo (ou variável) deve ser representado por um determinado *tipo de dado* que é característico da linguagem de programação sendo usada. Em linguagens de programação orientadas a objetos, é possível declarar campos como sendo referências à instâncias de outras classes já existentes.

Antes de ver como campos podem ser declarados e usados em classes, vamos ver os tipos de dados que poderão ser usados para estes campos e algumas operações básicas que a linguagem permite.

### 2.3.1 Usando dados nativos de Java

Existem poucos tipos de dados nativos em Java, isto é, dados que são parte da linguagem e não são instâncias de outras classes. Cada tipo de dado tem características e limitações diferentes, que serão comentadas brevemente aqui. Mais detalhes sobre os tipos de dados e usos corretos e incorretos são mostrados no apêndice A.

`boolean`

O tipo de dado `boolean` em Java representa valores lógicos ou booleanos. Este tipo de dado assume um dos dois valores `true` ou `false` que são constantes pré-definidas do tipo booleano. Campos do tipo `boolean` são usados especialmente para representar estados (verdadeiro/falso) e em expressões lógicas e condicionais. Usos corretos e incorretos do tipo de dado `boolean` são mostrados na seção A.2.1 do apêndice A.

Existem quatro tipos numéricos que representam valores inteiros (sem casas decimais) em Java:

`byte`
- O tipo `byte`, que pode ser usado para representar valores entre `-128` e `127`, inclusive, ocupando um único byte na memória.

`short`
- O tipo `short`, que pode ser usado para representar valores entre `-32768` e `32767`, inclusive, ocupando dois bytes na memória.

`int`
- O tipo `int`, que pode ser usado para representar valores entre `-2147483648` a `2147483647`, inclusive, ocupando quatro bytes na memória.

`long`
- O tipo `long`, que pode ser usado para representar valores entre `-9223372036854775808` e `9223372036854775807`, inclusive, ocupando oito bytes na memória.

O tipo `char` também é um tipo numérico inteiro, mas é mais usado para representar caracteres Unicode entre `0` e `65535`, inclusive, usando dois bytes. Caracteres são representados entre aspas simples, e operações matemáticas básicas podem ser feitas com campos do tipo `char` como se fossem de outros tipos inteiros.

*char*

Existem dois tipos de dados que são usados para representar valores numéricos de ponto flutuante: `float`, que representa valores com precisão simples, entre `1.40129846432481707e-45` e `3.40282346638528860e+38`, usando quatro bytes na memória, e `double`, que representa valores com precisão dupla, entre `4.94065645841246544e-324` e `1.79769313486231570e+308`, usando oito bytes na memória.

*float*
*double*

É interessante notar que todos os tipos numéricos de Java preservam suas características e limitações independentemente do sistema operacional ou arquitetura de computador sendo usada - por exemplo, um `long` ocupará oito bytes em qualquer sistema operacional onde a máquina virtual Java seja executada.

A classe `String` é usada para representar cadeias de caracteres (strings). Strings não são dados nativos, sendo instâncias da classe `String`, mas considerando que seu uso é simples e intuitivo, e que muitos exemplos podem se beneficiar do uso de instâncias da classe, veremos instâncias de `String` de forma similar a campos de tipos nativos. Detalhes da classe `String` serão vistos no capítulo 12.

*Classe* `String`
*strings*

Algumas das operações básicas que podem ser feitas com os tipos nativos numéricos (isto é, `byte`, `short`, `int`, `long`, `float`, `double` e até mesmo `char`) são: soma (com o operador `+`), subtração (com o operador `-`), multiplicação (com o operador `*`), divisão (com o operador `/`) e extração do módulo ou resto da divisão (com o operador `%`). Valores também podem ter seus sinais invertidos (positivo para negativo e vice-versa) colocando-se o sinal de menos (`-`) antes de um valor, variável ou campo numérico.

*operador* `+`
*operador* `-`
*operador* `*`
*operador* `=`
*operador* `%`

Valores numéricos podem ser comparados com operadores que retornam um valor do tipo `boolean` que será `true` ou `false` dependendo do resultado da operação. Os operadores são:

- `<` (retorna `true` se o valor à esquerda do sinal for menor que o valor à direita) — *operador* `<`
- `>` (retorna `true` se o valor à esquerda do sinal for maior que o valor à direita) — *operador* `>`
- `<=` (retorna `true` se o valor à esquerda do sinal for menor ou igual ao valor à direita) — *operador* `<=`
- `>=` (retorna `true` se o valor à esquerda do sinal for maior ou igual ao valor à direita) — *operador* `>=`
- `==` (retorna `true` se o valor à esquerda do sinal for exatamente igual ao valor à direita) — *operador* `==`
- `!=` (retorna `true` se o valor à esquerda do sinal for diferente do valor à direita) — *operador* `!=`

Valores booleanos (variáveis, campos, ou resultados de operações) podem ser combinados com três operadores lógicos. O operador `&&` (`E` lógico) retorna `true` se e somente se todos os valores booleanos da operação forem iguais a `true` - desta forma, o resultado da operação `true && true && true` será `true` e o resultado de `true && false` será `false`. O operador `||` (`OU` lógico) retorna `true` se qualquer um dos valores booleanos da operação for igual a `true` - assim, o resultado da operação `false || false || true` será `true` e o resultado de `false || false` será `false`. Finalmente, o operador `!` (`NÃO` lógico) retorna o inverso do valor booleano da operação (transformando valores `true` em `false` e vice-versa).

*operador* `&&`
`E` *lógico*
*operador* `||`
`OU` *lógico*
*operador* `!`
`NÃO` *lógico*

A única operação com instâncias da classe String que será vista agora é a concatenação, feita com o sinal de mais (`+`). Strings não podem ser comparadas com os operadores `>`, `<`, `==`, etc. -

operadores para comparação de strings serão vistos na seção 12.2.

Detalhes sobre os tipos nativos (com exemplos de uso correto e incorreto) são discutidos no apêndice A. A tabela 2.2 apresenta de forma resumida os tipos de dados básicos de Java.

| Tipo | Faixa de valores | Notas |
|---|---|---|
| `boolean` | `true` ou `false` | Valor booleano, podendo assumir uma das constantes `true` (verdadeiro) ou `false` (falso). Veja a seção A.2.1 no apêndice A. |
| `char` | `0` a `65535` | Caracteres representados em 16 bits, com suporte multilíngue, podendo ser usados como tipos inteiros de 16 bits, com sinal. Veja a seção A.2.2 no apêndice A. |
| `byte` | `-128` a `127` | Inteiro de 8 bits de precisão, com sinal. Veja a seção A.2.3 no apêndice A. |
| `short` | `-32768` a `32767` | Inteiro de 16 bits de precisão, com sinal. Veja a seção A.2.4 no apêndice A. |
| `int` | `-2147483648` a `2147483647` | Inteiro de 32 bits de precisão, com sinal. Veja a seção A.2.5 no apêndice A. |
| `long` | `-9223372036854775808` a `9223372036854775807` | Inteiro de 64 bits de precisão, com sinal. Veja a seção A.2.6 no apêndice A. |
| `float` | `1.40129846432481707e-45` a `3.40282346638528860e+38` | Ponto flutuante de precisão simples, armazenados em 32 bits, com sinal. Veja a seção A.2.7 no apêndice A. |
| `double` | `4.94065645841246544e-324` a `1.79769313486231570e+308` | Ponto flutuante de precisão dupla, armazenados em 64 bits, com sinal. Veja a seção A.2.8 no apêndice A. |
| `String` | Tamanho limitado à memória disponível. | Cadeia de caracteres que usam dois bytes por caracter. Strings podem ser vazias (zero caracteres) e conter qualquer tipo de caracter. Veja a seção 12.2 no capítulo 12. |

Tabela 2.2: Tipos básicos de Java

### 2.3.2 Declarando campos em classes em Java

*declaração de campos*

A declaração dos campos nas classes em Java é simples: basta declarar o tipo de dado, seguido dos nomes dos campos que serão daquele tipo.

A listagem 2.2 mostra alguns exemplos de declaração de campos. Esta listagem corresponde ao modelo mostrado na seção 1.5.4, mas considerando somente os dados do modelo, e não suas operações.

Listagem 2.2: A classe `RegistroAcademicoSimples`.

```
1 /**
2  * A classe RegistroAcademicoSimples, que contém somente alguns campos que
3  * exemplificam as declarações de campos e tipos de dados em Java.
4  * Esta classe depende da classe Data para ser compilada com sucesso.
5  */
6 class RegistroAcademicoSimples // declaração da classe
```

```
7   {
8   /**
9    * Declaração dos campos da classe
10   */
11  String nomeDoAluno; // uma cadeia de caracteres para representar um nome
12  int númeroDeMatrícula; // pode representar números com até 9 dígitos !
13  Data dataDeNascimento = new Data(); // uma referência à uma instância da classe Data
14  boolean éBolsista;  // valor simples: sim ou não (true ou false)
15  short anoDeMatrícula; // um short basta para representar anos
16  /* Se houvessem métodos para esta classe, eles seriam declarados aqui dentro. */
17
18  } // fim da classe RegistroAcademicoSimples
```

Na listagem 2.2 vemos que para cada dado do modelo `RegistroAcademico` existe um campo correspondente na classe. Cada campo tem um tipo de dado associado, nativo ou referência à uma classe. Dois exemplos de campos que são referências à uma classe são os campos `nomeDoAluno` e `dataDeNascimento`, declarados nas linhas 11 e 13, respectivamente.

A listagem 2.3 mostra outro exemplo de declaração de campos em uma classe. Novamente, a classe `DataSemMetodos` somente contém declarações de campos, mas não dos métodos que devem ser criados nesta classe.

Listagem 2.3: A classe `DataSemMetodos`.

```
1  /**
2   * A classe DataSemMetodos, que contém somente alguns campos que exemplificam as
3   * declarações de campos e tipos de dados em Java.
4   */
5  class DataSemMetodos // declaração da classe
6    {
7    /**
8     * Declaração dos campos da classe
9     */
10   byte dia,mês; // tanto o dia quanto o mês podem ser representados por bytes
11   short ano; // o ano completo (quatro dígitos) deve ser representado por um short
12   /* Se houvessem métodos para esta classe, eles seriam declarados aqui dentro. */
13
14   } // fim da classe DataSemMetodos
```

Algumas regras e informações sobre declaração de campos em classes em Java são mostradas a seguir:

*regras para nomes de campos*

- Nomes de campos seguem quase todas as mesmas regras de nomes de classes: devem ser iniciados por uma letra (incluindo _ e `$`), devem ser compostos de uma única palavra (sem espaços, vírgulas, etc.) e podem ser compostos de letras e números. Ao contrário de nomes de classes, nomes de campos podem conter acentos. Nomes de campos não podem ser iguais a nenhuma das palavras reservadas mostradas na tabela 2.1.

*convenções para nomes de campos*

- Tradicionalmente nomes de campos e variáveis começam com caracteres minúsculos, alternando com maiúsculos entre palavras. Alguns exemplos de campos que seguem esta convenção são `dia`, `nomeDoAluno`, `rotaçõesPorMinuto` e `estadoDaLâmpada`.
- Campos podem ser declarados com a forma `modificador-de-acesso tipo-ou-classe nome-do-campo;` (terminado com ponto-e-vírgula) como nas linhas 11 a 15 da listagem 2.2. Modificadores de acesso serão vistos na seção 2.6, mas nos exemplos a sua declaração foi omitida propositadamente. Se vários campos forem do mesmo tipo, podemos declará-los simultaneamente com a forma `modificador-de-acesso tipo-ou-classe`

`nome-do-campo1, nome-do-campo2, nome-do-campo3;` (nomes separados por vírgulas, declaração terminada com ponto-e-vírgula) como na linha 10 da listagem 2.3. É aconselhável evitar a declaração de muitos campos em uma mesma linha de programa para não comprometer a clareza do código.

- Se o campo for referência à instância de uma classe, será necessário inicializar a referência com a palavra-chave `new` . O formato padrão de inicialização de referências é `NomeDaClasse nomeDaReferência = new NomeDaClasse();` (com ponto-e-vírgula no final), como mostrado na linha 13 da listagem 2.2. É possível somente declarar a referência e inicializá-la depois em algum método da classe, assim como passar valores para a inicialização das instâncias, o que será visto no capítulo 4.

*new*

- Dois campos da mesma classe não podem ter exatamente o mesmo nome. Por exemplo, não podemos declarar outro campo chamado `dia` na classe da listagem 2.3 mesmo se o tipo de dado for diferente.

### 2.3.3 Que tipos de dados devem ser usados ?

*escolha do tipo de dado adequado*

Ao escolher que tipos de dados ou classes devem ser usados para que campos, o programador das classes deve considerar em especial a faixa de valores que o campo pode vir a ocupar. Por exemplo, o campo `númeroDeMatrícula` na classe da listagem 2.2 poderia ser do tipo `long`, mas dificilmente teríamos necessidade de valores para números de matrícula acima de nove dígitos (que podem perfeitamente ser representados por um `int`). Similarmente, o tipo `short` poderia ser usado, mas somente números de matrícula entre 0 e 32767 poderiam ser representados (considerando que não devem existir números de matrícula negativos).

Programadores inexperientes podem cair na tentação de usar somente um ou dois tipos de dados, por exemplo, considerando que qualquer valor numérico pode ser representado por um tipo `long` ou `double`, pensando que a faixa de valores representáveis por estes tipos é grande o suficiente para poder representar qualquer valor.

A escolha de tipos inadequados para os campos pode ser dispendiosa, criar complexidades desnecessárias e até mesmo criar erros de lógica de programação: se, por exemplo, o campo `dia` da classe `DataSemMetodos` (listagem 2.3) for do tipo `long`, oito bytes estarão sendo usados onde somente um bastaria. Se, para representar um número de registro de aluno, resolvermos usar uma instância da classe `String` ao invés de um inteiro, além do uso desnecessário de memória para representar os caracteres na string, a tarefa de procurar o último (ou maior) número de matrícula de uma turma seria mais complexa que o necessário, já que a comparação entre strings é computacionalmente mais custosa do que entre inteiros.

Por outro lado, economia de bytes em excesso pode ser prejudicial - consideremos o caso do campo `ano` no modelo `DataSemMetodos` (listagem 2.3), que é representado por um valor do tipo `short`, que pode representar valores positivos até 32768. Pode-se argumentar que não existe nenhuma aplicação que hoje necessite calcular ou representar datas mais de 30.000 anos no futuro, mas a alternativa seria usar um valor do tipo byte (valor máximo 127) e acrescentar uma constante quando fosse necessário. Esta constante seria o ano mínimo representável pela classe, que poderia ser, por exemplo, 1970. Desta forma, quando o campo contivesse o valor 31, na verdade o ano representado seria (1970+31) = 2001.

Usando esta técnica e um byte para representar o ano, a classe poderia representar anos até (1970+127) = 2097, o que deve ser aceitável para a maioria das aplicações, mas a complexidade da representação só nos faz economizar um único byte, o que realmente não vale a pena.

Valores de tipos numéricos nativos podem ser transformados em outros tipos numéricos também nativos através de uma operação conhecida como *cast* ou conversão explícita. Para efetuar a conversão explícita, basta colocar o tipo de dado para o qual se deseja converter entre parênteses antes da expressão a ser convertida (ex.: o valor `(byte)100` será considerado do tipo `byte`, e o valor `(double)100;` do tipo `double`).

*cast*
*conversão de tipos*

A necessidade do uso do operador de conversão explícita será demonstrado em capítulos posteriores. Problemas no uso do *cast* são discutidos nos exemplos do apêndice A.

## 2.4 Métodos em classes em Java

Até agora foi mostrado como podemos criar classes e representar campos nestas classes. A maioria das classes representa modelos que tem dados e operações que manipulam estes dados. As operações dos modelos, que em Programação Orientada a Objetos são conhecidas como métodos, também seguem regras rígidas para sua criação, descritas a seguir:

*declaração de métodos*

- Nomes de métodos seguem as mesmas regras de nomes de campos: devem ser iniciados por uma letra (incluindo _ e `$`), devem ser compostos de uma única palavra (sem espaços, vírgulas, etc.), podem ser compostos de letras e números e podem conter acentos. Nomes de métodos não podem ser iguais a nenhuma das palavras reservadas mostradas na tabela 2.1.

*regras para nomes de métodos*

- Nomes de métodos refletem ações que são efetuadas nos campos da classe e/ou valores passados como argumentos para estes métodos. Embora esta não seja uma regra obrigatória, se for seguida os programas e classes poderão ser lidos e compreendidos mais facilmente por programadores. Tradicionalmente os nomes de métodos começam com caracteres minúsculos e alternam para maiúsculos entre palavras. Alguns exemplos de nomes de métodos que seguem esta regra são `mostraDados`, `acende` e `inicializaData`.

*convenções para nomes de métodos*

- Métodos não podem ser criados dentro de outros métodos, nem fora da classe a qual pertencem - não podemos ter métodos isolados, fora de todas as classes.

Para exemplificar as regras de criação de métodos, veremos a classe `DataSimples`, com campos e métodos básicos, mostrada na listagem 2.4.

Listagem 2.4: A classe `DataSimples`.

```
1  /**
2   * A classe DataSimples, que contém campos e métodos que permitem a manipulação de
3   * datas.
4   */
5  class DataSimples // declaração da classe
6    {
7    /**
8     * Declaração dos campos da classe
9     */
10   byte dia,mês; // dia e mês são representados por bytes
11   short ano; // ano é representado por um short
12
13   /**
14    * O método inicializaDataSimples recebe argumentos para inicializar os campos da
15    * classe DataSimples. Este método chama o método dataÉVálida para verificar se os
16    * argumentos são correspondentes a uma data válida: se forem, inicializa os
17    * campos, caso contrário inicializa todos os três campos com o valor zero.
18    * @param d o argumento correspondente ao método dia
19    * @param m o argumento correspondente ao método mês
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
20    * @param a o argumento correspondente ao método ano
21    */
22   void inicializaDataSimples(byte d,byte m,short a)
23     {
24     if (dataÉVálida(d,m,a)) // se a data for válida, inicializa os campos com os
25                             // valores passados como argumentos
26       {
27       dia = d; mês = m; ano = a;
28       }
29     else // caso contrário, inicializa os campos com zero
30       {
31       dia = 0; mês = 0; ano = 0;
32       }
33     } // fim do método inicializaDataSimples
34
35   /**
36    * O método dataÉVálida recebe três valores como argumentos e verifica de maneira
37    * simples se os dados correspondem a uma data válida. Se a data for válida, retorna
38    * a constante booleana true, caso contrário, retorna a constante booleana false.
39    * Vale a pena notar que este algoritmo é simples e incorreto, um dos exercícios
40    * sugere a implementação do algoritmo correto.
41    * @param d o argumento correspondente ao método dia
42    * @param m o argumento correspondente ao método mês
43    * @param a o argumento correspondente ao método ano
44    * @return true se a data for válida, false se não for válida
45    */
46   boolean dataÉVálida(byte d,byte m,short a)
47     {
48     if ((d >=1) &&      // se o dia for maior ou igual a 1  E
49         (d <= 31) &&    // se o dia for menor ou igual a 31  E
50         (m >= 1) &&     // se o mês for maior ou igual a 1  E
51         (m <= 12))      // se o mês for menor ou igual a 12 ENTÃO
52       return true;      // a data é válida, retorna true
53     else
54       return false;     // a data não é válida, retorna false
55     } // fim do método dataÉVálida
56
57   /**
58    * O método éIgual recebe uma instância da própria classe DataSimples como argumento
59    * e verifica se a data representada pela classe e pela instância que foi passada é
60    * a mesma. A comparação é feita comparando os campos da data um a um.
61    * @param outraDataSimples uma instância da própria classe DataSimples
62    * @return true se a data encapsulada for igual à passada, false caso contrário
63    */
64   boolean éIgual(DataSimples outraDataSimples)
65     {
66     if ((dia == outraDataSimples.dia) &&    // se os dois dias forem iguais E
67         (mês == outraDataSimples.mês) &&    // se os dois meses forem iguais E
68         (ano == outraDataSimples.ano))      // se os dois anos forem iguais então
69       return true;      // a data é igual, retorna true
70     else
71       return false;     // a data é diferente, retorna false
72     } // fim do método éIgual
73
74   /**
75    * O método mostraDataSimples não recebe argumentos nem retorna valores. Este método
76    * somente imprime os valores dos campos, formatados de forma que uma barra ("/")
77    * seja impressa entre eles. Quando o valor do ano for impresso, uma quebra de
78    * linha também será impressa.
79    */
80   void mostraDataSimples()
81     {
82     System.out.print(dia);   // O método print do campo out da classe System faz com
83     System.out.print("/");   // que o argumento passado a ele seja transformado em uma
84     System.out.print(mês);   // String e impresso no terminal. O método println faz a
85     System.out.print("/");   // mesma coisa, mas adiciona uma quebra de linha ('\n')
86     System.out.println(ano); // ao final da String impressa.
87     }  // fim do método mostraDataSimples
88
```

```
89  } // fim da classe DataSimples
```

Usando a listagem 2.4 como exemplo, podemos ver outras regras sobre declaração de métodos em classes em Java:

- O formato genérico usado para a declaração de métodos é `modificador-de-acesso tipo-ou-classe-de-retorno nome-do-método(lista-de-argumentos)`. A declaração de métodos **não** é terminada com ponto-e-vírgula. Modificadores de acesso serão vistos na seção 2.6, na listagem a declaração destes modificadores foi omitida propositadamente.
- Cada método deve ter, na sua declaração, um tipo ou classe de retorno, correspondente ao valor que o método deve retornar. Caso o método não retorne nada, isto é, caso ele somente execute alguma operação sem precisar retornar valores, o valor de retorno deverá ser `void`, um tipo de retorno especial que indica que o retorno deve ser desconsiderado. Na classe `Data` (listagem 2.4), os métodos `inicializaDataSimples` e `mostraDataSimples`, respectivamente declarados nas linhas 22 e 80, retornam `void`[3], enquanto os métodos `dataÉVálida` e `éIgual` (linhas 46 e 64) retornam um valor do tipo `boolean`. Métodos podem retornar também instâncias de classes, mas somente um valor pode ser retornado de cada vez.

  *retorno de métodos* `void`
- Métodos que retornam algum valor diferente de `void` devem ter, em seu corpo, a palavra-chave `return` seguida de uma constante ou variável do tipo ou classe que foi declarada como sendo a de retorno do método. Na classe `Data` (listagem 2.4) o método `dataÉVálida` (linha 46) declara que deverá retornar um valor do tipo booleano, o que será feito através da palavra-chave `return` na linha 52 ou na linha 54, dependendo da expressão condicional na linha 48. Métodos que retornam `void` não precisam ter a palavra-chave `return` no seu corpo, e se tiverem, esta não deve ser seguida de nenhuma constante ou variável.

  `return`
- Métodos podem ter listas de argumentos, ou seja, variáveis contendo valores que podem ser usados pelos métodos para efetuar suas operações. Na listagem 2.4, o método `inicializaDataSimples` (linha 22) recebe três argumentos, e o método `éIgual` (linha 64) recebe como argumento uma instância da própria classe `Data`. Cada argumento passado para um método deve ter uma referência ou nome de variável correspondente, e cada argumento deve ser precedido de seu tipo ou classe. Opcionalmente, métodos podem não receber argumentos, como é o caso do método `mostraDataSimples` na mesma listagem.

  *argumentos ou parâmetros para métodos*

Um ponto interessante da listagem 2.4 é o método `éIgual`, que recebe como argumento uma instância da própria classe `Data`, para comparação de igualdade. Dentro deste método, os campos da classe `dia`, `mês` e `ano` são comparados, respectivamente, com os mesmos campos da instância que foi passada como argumento.

## 2.5 Escopo

O *escopo* dos campos e variáveis dentro de uma classe determina a sua visibilidade (isto é, se as variáveis ou campos podem ser acessadas ou modificadas em todos os métodos da classe, somente um determinado método ou mesmo somente em parte de um determinado método). A classe `Triangulo`, mostrada na listagem 2.5, será usada para exemplificar alguns pontos sobre escopo de campos e variáveis.

*escopo de variáveis e campos*

[3] Apesar do valor e tipo de retorno serem desconsiderados quando usamos `void`, e de não existir nenhum comando no método que o faça retornar um valor, é comum dizer que um método "retorna `void`".

Listagem 2.5: A classe `Triangulo`.

```
1  /**
2   * A classe Triangulo, que representa os três lados de um triângulo qualquer.
3   */
4  class Triangulo // declaração da classe
5    {
6    /**
7     * Declaração de um dos campos da classe
8     */
9    float lado1;
10
11   /**
12    * O método éEquilátero verifica se o triângulo é equilátero ou não.
13    * @return true se o triângulo é equilátero, false se não for
14    */
15   boolean éEquilátero()
16     {
17     boolean igualdade12,resultado;
18     igualdade12 = (lado1 == lado2); // o lado 1 é igual ao lado 2 ?
19     boolean igualdade23;
20     igualdade23 = (lado2 == lado3); // o lado 2 é igual ao lado 3 ?
21     if (igualdade12 && igualdade23) // os três lados são iguais ?
22       resultado = true;
23     else
24       resultado = false;
25     return resultado;
26     } // fim do método éEquilátero
27
28   /**
29    * O método calculaPerímetro calcula o perímetro do triângulo usando seus lados.
30    * @return o perímetro do triângulo representado por esta classe
31    */
32   float calculaPerímetro()
33     {
34     float resultado = lado1+lado2+lado3;
35     return resultado;
36     } // fim do método perímetro
37
38   /**
39    * Declaração dos outros campos da classe
40    */
41   float lado2,lado3;
42
43   } // fim da classe Triangulo
```

Algumas regras simples de escopo são mostradas a seguir:

*escopo de campos de uma classe*

- Campos declarados em uma classe são válidos por toda a classe, isto é, o escopo de campos de uma classe é toda a classe, *mesmo que os campos estejam declarados depois dos métodos que os usam*, como mostrado na listagem 2.5, onde os campos `lado1`, `lado2` e `lado3` podem ser usados nos métodos `éEquilátero` e `calculaPerímetro` mesmo "antes" (no sentido de leitura da listagem, de cima para baixo) de serem declarados. Vale a pena notar que para o compilador Java, a ordem em que as declarações de métodos e campos aparecem é irrelevante.

*escopo de variáveis em um método*

- Variáveis e instâncias declaradas dentro de métodos só serão válidas dentro destes métodos. Desta forma, as variáveis `igualdade12` e `igualdade23`, declaradas no método `éEquilátero`, somente serão válidas dentro deste método. Outras variáveis podem ser declaradas em outros métodos, mas mesmo que tenham o mesmo nome serão consideradas variáveis diferentes e não relacionadas, como por exemplo, a variável `resultado`, que foi declarada como sendo do tipo `boolean` no método `éEquilátero` e como `float` no método `calculaPerímetro`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

- Dentro de métodos e blocos de comandos, a ordem de declaração de variáveis e referências à instâncias é considerada: se as linhas 20 e 21 da listagem 2.5 fossem trocadas, um erro de compilação ocorreria pois o compilador não aceitaria que a variável `igualdade23` recebesse um valor antes de ser declarada. Em resumo, variáveis dentro de métodos só podem ser usadas depois de declaradas.
- Variáveis passadas como argumentos para os métodos só são válidas dentro dos métodos. Por exemplo, as variáveis `d`, `m` e `a`, que são passadas como argumentos para o método `dataÉVálida` na classe `Data` (listagem 2.4) **não** são visíveis no método `imprimeData` da mesma classe.

## 2.6 Modificadores de acesso

Uma das principais vantagens do paradigma de orientação a objetos é a possibilidade de encapsular campos e métodos capazes de manipular estes campos em uma classe, conforme comentado na seção 1.4. É desejável que os campos das classes sejam ocultos ou escondidos dos programadores usuários das classes, para evitar que os dados sejam manipulados diretamente ao invés de através dos métodos das classes.

Os exemplos mostrados até agora demonstram o encapsulamento de campos e métodos, mas sem fazer com que os dados sejam ocultos - os programadores usuários podem manipular diretamente os campos, como é demonstrado na classe da listagem 2.6.

Listagem 2.6: A classe `DemoDataSimples`, que pode ser executada e que demonstra o uso de instâncias da classe `DataSimples`.

```
1  /**
2   * A classe DemoDataSimples, que demonstra usos da classe DataSimples, em especial os
3   * problemas que podem ocorrer quando os campos de uma classe podem ser acessados
4   * diretamente.
5   */
6  class DemoDataSimples // declaração da classe
7    {
8    /**
9     * O método main permite a execução desta classe. Este método contém declarações
10    * de algumas instâncias da classe DataSimples, e demonstra como seus campos podem
11    * ser acessados diretamente pois estes são públicos.
12    * @param argumentos os argumentos que podem ser passados para o método via linha
13    *        de comando, mas que neste caso serão ignorados.
14    */
15   public static void main(String[] argumentos)
16     {
17     // Criamos duas instâncias da classe DataSimples, usando a palavra-chave new. As
18     // instâncias serão associadas à duas referências, que permitirão o acesso aos
19     // campos e métodos das instâncias.
20     DataSimples hoje = new DataSimples();
21     DataSimples independênciaDoBrasil = new DataSimples();
22     // E três variáveis para receber o dia, mês e ano para as datas
23     byte umDia,umMês; short umAno;
24     // Inicializamos "hoje" com uma data não-válida
25     umDia = 40; umMês = 1; umAno = 2001;
26     hoje.inicializaDataSimples(umDia,umMês,umAno); // inicializa os campos da instância
27     hoje.mostraDataSimples();  // imprime 0/0/0
28     // Inicializamos "independênciaDoBrasil" com uma data válida
29     umDia = 7; umMês = 9; umAno = 1822;
30     independênciaDoBrasil.inicializaDataSimples(umDia,umMês,umAno);
31     independênciaDoBrasil.mostraDataSimples(); // imprime 7/9/1822
32     // Vamos testar o método éIgual:
33     if (hoje.éIgual(independênciaDoBrasil))
34       System.out.println("As datas são iguais !");
35     else
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
36        System.out.println("As datas são diferentes !");
37      // O problema: podemos facilmente "invalidar" datas válidas acessando os seus
38      // campos diretamente:
39      hoje.dia = 0;
40      hoje.mês = 1;
41      hoje.ano = 2001;
42      hoje.mostraDataSimples(); // imprime 0/1/2001 - é válida ou não ?
43      independênciaDoBrasil.mês = 13;
44      independênciaDoBrasil.mostraDataSimples(); // imprime 7/13/1822 - é válida ou não ?
45      }  // fim do método main
46
47  } // fim da classe DemoDataSimples
```

A classe na listagem 2.6 contém o método `main`, que permite a execução da classe. Mais informações sobre classes executáveis e o método `main` serão vistas no capítulo 3. Por enquanto, assumiremos que o conteúdo do método `main` será executado passo a passo pela máquina virtual Java.

Alguns dos pontos de interesse na listagem 2.6 são:

- Duas instâncias da classe `DataSimples` e duas referências a estas instâncias são declaradas e alocadas simultaneamente nas linhas 20 e 21 da listagem 2.6. Após a inicialização, as referências podem acessar os campos e métodos da classe `DataSimples`.
- Para receber os valores correspondentes ao dia, mês e ano, três variáveis de tipos compatíveis com os definidos dentro da classe `DataSimples` são declarados na linha 23. Como estas variáveis são de tipos nativos, **não** devem ser inicializadas com a palavra-chave `new` - podem ser usadas diretamente após a declaração.

*executando métodos*

- Métodos que foram definidos dentro das classes podem ser executados usando a notação `nome-da-referência.nome-do-método(argumentos);`, mas somente se a instância correspondente tiver sido inicializada com a palavra-chave `new`. O acesso aos métodos de uma classe pode ser feito usando-se o operador ponto (`.`), que separa o nome da referência do nome do método. Se houver argumentos para o método estes deverão ser passados entre parênteses e separados por vírgula, mas **sem** os tipos de dados ou, se os argumentos forem instâncias de classes, sem os nomes das classes. Se não houverem argumentos, parênteses sem conteúdo devem ser usados, como na linhas 31 e 44 da listagem 2.6.

*operador ponto (.)*

*mensagens*

- A execução de métodos também é conhecida como *envio de mensagens* a objetos: na linha 26 da listagem 2.6, a classe em Java que está sendo executada está enviando a mensagem `inicializaDataSimples` ao objeto `hoje` com os argumentos `umDia`, `umMês` e `umAno`.

*acessando campos*

- Campos de classes também podem ser acessados a partir de referências à instâncias das classes, usando o operador ponto com a notação `nome-da-referência.nome-do-campo;` como demonstrado nas linhas 30 a 41 e 43.

O problema demonstrado na listagem 2.6 é que não podemos mais garantir que datas não-válidas serão processadas adequadamente pela classe. Um dos objetivos ao fazer o encapsulamento dos dados de uma data em uma classe era garantir que, caso valores inadequados fossem passados para os campos `dia`, `mês` e `ano`, os três campos receberiam o valor zero, sinalando que a data não era válida. Como os campos podem ser manipulados diretamente usando a notação do ponto, podemos criar uma data válida e posteriomente modificar somente um dos campos de forma que a data fique não-válida, apesar de alguns campos serem diferentes de zero, como mostrado nas linhas 39 e 43.

*restrição ao acesso a campos*

O ideal então seria restringir o acesso aos campos para que o programador usuário não possa

acessá-los diretamente - os campos somente poderiam ser modificados através dos métodos, garantindo que, no exemplo, datas não-válidas tenham todos os seus campos valendo zero. Java permite a restrição ao acesso a campos e métodos em classes através de *modificadores de acesso* que são declarados dentro das classes, antes dos métodos e campos. Existem quatro modificadores de acesso, descritos a seguir.

*modificador* `public`

O modificador `public` garante que o campo ou método da classe declarado com este modificador poderá ser acessado ou executado a partir de qualquer outra classe. Campos e métodos que devam ser acessados (e modificados, no caso de campos) devem ser declarados com o modificador `public`.

*modificador* `private`

Campos e métodos declarados com o modificador `private` só podem ser acessados, modificados ou executados por métodos da mesma classe, sendo completamente ocultos para o programador usuário que for usar instâncias desta classe ou criar classes herdeiras ou derivadas (que serão vistas no capítulo 8). Campos ou métodos que devam ser ocultos totalmente de usuários da classe devem ser declarados com este modificador.

*modificador* `protected`

O modificador `protected` funciona como o modificador `private` exceto que classes herdeiras ou derivadas também terão acesso ao campo ou método marcado com este modificador. O comportamento deste modificador será visto na seção 10.3.2.

*package* *friendly*

Finalmente, campos e métodos podem ser declarados sem modificadores. Neste caso, eles serão considerados como tendo sendo da categoria *package* ou *friendly*, significando que seus campos e métodos serão visíveis (podendo ser acessados) para todas as classes pertencentes a um mesmo pacote ou *package*. Pacotes de classes serão vistos no capítulo 10, e o papel dos modificadores nos pacotes na seção 10.3.

Ao criar classes, o programador de classes deve implementar uma política de ocultação ou de acesso a dados e métodos internos (que não devam ser acessados por programadores usuários). Algumas regras básicas para implementação de políticas para classes simples em Java são:

1. Todos os campos de uma classe devem ser declarados com o modificador `private` ou com o modificador `protected`. As diferenças entre os dois modificadores serão vistas na seção 10.3.2. Campos declarados desta forma estarão ocultos para o programador usuário das classes.
2. Métodos de uma classe que devem ser acessíveis (isto é, não devem ser ocultos) devem ser declarados explicitamente com o modificador `public`. A omissão da declaração do modificador não trará problemas aparentes, uma vez que o método será considerado como pertencente à categoria *package*, mas caso classes desenvolvidas venham a ser agrupadas em pacotes, problemas de acesso poderão ocorrer. A declaração explícita do modificador `public` para os métodos evita esses problemas.
3. Como em princípio os campos terão o modificador `private`, métodos que permitam a manipulação controlada dos valores dos campos devem ser escritos nas classes, e estes métodos devem ter o modificador `public`.
4. Se for necessário ou desejável, métodos de uma classe podem ser declarados com o modificador `private` - estes métodos não poderão ser executados por classes escritas por programadores-usuários, mas poderão ser executados por outros métodos dentro da mesma classe. Um exemplo hipotético seria o método `imprimeNúmeroDaPágina` em uma classe `Relatório`, que tem o método `imprimeRelatório`. O método `imprimeRelatório` poderia, sempre que necessário, executar o método `imprimeNúmeroDaPágina`, mas raramente



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

seria adequado ao programador usuário executar diretamente este método, que poderia ser declarado como `private`.

Evidentemente estas regras cobrem somente os casos básicos de criação de classes. Mecanismos de herança e criação de pacotes de classes terão necessidades e características que farão com que revisemos os conceitos de modificadores.

A classe `Data`, mostrada na listagem 2.7, encapsula os valores para representar uma data fazendo com que os dados encapsulados só possam ser acessados via seus métodos.

Listagem 2.7: A classe `Data`, que implementa uma política de ocultação de campos.

```
1  /**
2   * A classe Data, que contém campos e métodos que permitem a manipulação de datas.
3   * Esta versão da classe tem os campos protegidos para evitar a sua manipulação
4   * incorreta - valores dos campos só podem ser acessados através de métodos.
5   */
6  class Data // declaração da classe
7    {
8    /**
9     * Declaração dos campos da classe
10    */
11   private byte dia,mês; // dia e mês são representados por bytes
12   private short ano; // ano é representado por um short
13
14   /**
15    * O método inicializaData recebe argumentos para inicializar os campos da classe
16    * Data. Este método chama o método dataÉVálida para verificar se os argumentos
17    * correspondem a uma data válida: se forem, inicializa os campos, caso contrário
18    * inicializa todos os três campos com o valor zero.
19    * @param d o argumento correspondente ao método dia
20    * @param m o argumento correspondente ao método mês
21    * @param a o argumento correspondente ao método ano
22    */
23   public void inicializaData(byte d,byte m,short a)
24     {
25     if (dataÉVálida(d,m,a)) // se a data for válida, inicializa os campos com os
26                             // valores passados como argumentos
27       {
28       dia = d; mês = m; ano = a;
29       }
30     else // caso contrário, inicializa os campos com zero
31       {
32       dia = 0; mês = 0; ano = 0;
33       }
34     } // fim do método inicializaData
35
36   /**
37    * O método retornaDia retorna o dia da data encapsulada, já que o campo dia, sendo
38    * declarado como private, não poderá ser acessado diretamente de fora da classe.
39    * @return o dia encapsulado pela classe
40    */
41   public byte retornaDia()
42     {
43     return dia;
44     } // fim do método retornaDia
45
46   /**
47    * O método retornaMês retorna o mês da data encapsulada, já que o campo mês, sendo
48    * declarado como private, não poderá ser acessado diretamente de fora da classe.
49    * @return o mês encapsulado pela classe
50    */
51   public byte retornaMês()
52     {
```

```
53      return mês;
54      } // fim do método retornaMês
55
56    /**
57     * O método retornaAno retorna o ano da data encapsulada, já que o campo ano, sendo
58     * declarado como private, não poderá ser acessado diretamente de fora da classe.
59     * @return o ano encapsulado pela classe
60     */
61    public short retornaAno()
62      {
63      return ano;
64      } // fim do método retornaAno
65
66    /**
67     * O método dataÉVálida recebe três valores como argumentos e verifica de maneira
68     * simples se os dados correspondem a uma data válida. Se a data for válida, retorna
69     * a constante booleana true, caso contrário, retorna a constante booleana false.
70     * Vale a pena notar que este algoritmo é simples e incorreto, um dos exercícios
71     * sugere a implementação do algoritmo correto.
72     * @param d o argumento correspondente ao método dia
73     * @param m o argumento correspondente ao método mês
74     * @param a o argumento correspondente ao método ano
75     * @return true se a data for válida, false se não for válida
76     */
77    public boolean dataÉVálida(byte d,byte m,short a)
78      {
79      if ((d >=1) &&       // se o dia for maior ou igual a 1  E
80          (d <= 31) &&     // se o dia for menor ou igual a 31  E
81          (m >= 1) &&      // se o mês for maior ou igual a 1  E
82          (m <= 12))       // se o mês for menor ou igual a 12  ENTÃO
83        return true;       // a data é válida, retorna true
84      else
85        return false;      // a data não é válida, retorna false
86      } // fim do método dataÉVálida
87
88    /**
89     * O método éIgual recebe uma instância da própria classe Data como argumento e
90     * verifica se a data representada pela classe e pela instância que foi passada é
91     * a mesma. A comparação é feita comparando os campos da data um a um.
92     * @param outraData uma instância da própria classe Data
93     * @return true se a data encapsulada for igual à passada, false caso contrário
94     */
95    public boolean éIgual(Data outraData)
96      {
97      if ((dia == outraData.dia) &&     // se os dois dias forem iguais E
98          (mês == outraData.mês) &&     // se os dois meses forem iguais E
99          (ano == outraData.ano))       // se os dois anos forem iguais então
100       return true;       // a data é igual, retorna true
101     else
102       return false;      // a data é diferente, retorna false
103     } // fim do método éIgual
104
105   /**
106    * O método mostraData não recebe argumentos nem retorna valores. Este método somente
107    * imprime os valores dos campos, formatados de forma que uma barra ("/") seja
108    * impressa entre eles. Quando o valor do ano for impresso, uma quebra de linha
109    * também será impressa.
110    */
111   public void mostraData()
112     {
113     System.out.print(dia);
114     System.out.print("/");
115     System.out.print(mês);
116     System.out.print("/");
117     System.out.println(ano);
118     }  // fim do método mostraData
119
120   } // fim da classe Data
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

A classe `Data` tem todos os seus campos declarados como `private` e métodos declarados como `public`, significando que os campos somente poderão ser acessados e modificados através dos métodos adequados (no caso, `inicializaData` para modificar os campos e `retornaDia`, `retornaMês`, `retornaAno` e `mostraData` para recuperar os valores dos campos).

O método `éIgual` da classe `Data` parece contradizer o que foi dito em relação ao modificador `private` - nas linhas 97 a 99 da listagem 2.7 os campos da instância `outraData` da classe `Data` são acessados diretamente apesar de terem sido declarados como `private` nas linhas 11 e 12 da mesma listagem. A listagem não está incorreta: o modificador `private` faz com que os campos e métodos modificados só possam ser acessados por métodos da mesma classe. Como a classe que implementa o método `éIgual` é exatamente a mesma da instância do argumento, ela tem acesso aos campos e métodos privados de si mesma.

A classe `DemoData`, mostrada na listagem 2.8, demonstra o uso de instâncias da classe `Data`. A tentativa de acesso a dados privados da classe `Data` causará erros de compilação nas linhas 40 e 41 da listagem 2.8 - os campos estão efetivamente protegidos contra tentativas de acesso direto.

Listagem 2.8: A classe `DemoData`, que pode ser executada e que demonstra o uso de instâncias da classe `Data`.

```
1  /**
2   * A classe DemoData, que demonstra usos da classe Data, em especial os efeitos
3   * causados pela modificação dos modificadores de acesso dos campos.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoData // declaração da classe
7    {
8    /**
9     * O método main permite a execução desta classe. Este método contém declarações
10    * de algumas instâncias da classe Data, e demonstra como usar a classe e como
11    * seus campos não podem ser acessados diretamente por ser privados.
12    * @param argumentos os argumentos que podem ser passados para o método via linha
13    *        de comando, mas que neste caso serão ignorados.
14    */
15   public static void main(String[] argumentos)
16     {
17     // Criamos duas instâncias da classe Data, usando a palavra-chave new. As
18     // instâncias serão associadas à duas referências, que permitirão o acesso aos
19     // métodos das instâncias mas não a seus campos.
20     Data nascimento = new Data();
21     Data descobrimentoDoBrasil = new Data();
22     // E três variáveis para receber o dia, mês e ano para as datas
23     byte umDia,umMês; short umAno;
24     // Inicializamos "nascimento" com uma data não-válida
25     umDia = 22; umMês = 13; umAno = 2000;
26     nascimento.inicializaData(umDia,umMês,umAno); // inicializa os campos da instância
27     nascimento.mostraData();  // imprime 0/0/0
28     // Inicializamos "descobrimentoDoBrasil" com uma data válida
29     umDia = 22; umMês = 4; umAno = 1500;
30     descobrimentoDoBrasil.inicializaData(umDia,umMês,umAno); // inicializa os campos
31                                                              // da instância
32     descobrimentoDoBrasil.mostraData(); // imprime 22/4/1500
33     // As datas são iguais ?
34     if (nascimento.éIgual(descobrimentoDoBrasil))
35       System.out.println("As datas são iguais !");
36     else
37       System.out.println("As datas são diferentes !");
38     // Como os campos são privados, não podemos mais acessá-los nem modificá-los.
39     // As duas linhas abaixo causarão erros de compilação !
40     nascimento.mês = 10;
41     descobrimentoDoBrasil.mês = -1;
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
42     }  // fim do método main
43
44   } // fim da classe DemoData
```

## 2.7 Exercícios do capítulo 2

Para os exercícios relacionados com criação de classes, é sugerido que se use os modificadores de acesso adequados a cada campo e método, tentando fazer com que os campos sejam sempre protegidos para acesso e que os métodos sejam públicos quando devido. Para isso, deve-se tentar escrever ao menos métodos para inicializar ou modificar os campos e imprimir os valores destes.

Para praticar, também é sugerida a criação também dos diagramas mostrados nas seções 1.5.1 a 1.5.4 e a inclusão de comentários nas classes, métodos e trechos adequados.

**Exercício 2.1:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** four
**B.** for
**C.** from
**D.** 4
**E.** FOR

**Exercício 2.2:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** do
**B.** DO
**C.** dont
**D.** don't
**E.** dodo
**F.** d*o

**Exercício 2.3:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** nome
**B.** nome!
**C.** nOmE
**D.** um_nome
**E.** nome(1)
**F.** newnome
**G.** nome___
**H.** nome$1

**Exercício 2.4:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** contador
**B.** 1contador
**C.** contador de linhas
**D.** Contador
**E.** count
**F.** counter

**Exercício 2.5:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** class
**B.** classe
**C.** Class
**D.** class2
**E.** 2class

**Exercício 2.6:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** new
**B.** novo
**C.** new1
**D.** 1new
**E.** novo_new

**Exercício 2.7:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** static
**B.** statico
**C.** estatico
**D.** estático
**E.** ESTÁTICO

**Exercício 2.8:** ⋆

Quais dos identificadores abaixo podem ser usados como nomes de classes, campos, métodos e variáveis em Java ? Quais não podem, e porquê ?

**A.** dia&noite
**B.** diaENoite
**C.** dia & noite
**D.** dia E noite
**E.** dia_e_noite

**Exercício 2.9:** ⋆

Responda verdadeiro ou falso para cada uma das afirmações abaixo, explicando ou justificando a sua resposta.

**A.** Um valor do tipo `boolean` pode receber o valor numérico zero.
**B.** Valores do tipo `short` podem estar entre 0 e $2^{16} - 1$.
**C.** Um valor do tipo `float` pode armazenar valores maiores do que os que podem ser armazenados por um valor do tipo `long`.
**D.** Podemos ter caracteres cujos valores sejam negativos.
**E.** O número de bytes ocupados por uma variável do tipo `float` depende do computador e do sistema operacional sendo usado.
**F.** Valores do tipo `byte` podem estar entre $-2^7$ e $2^7 - 1$.
**G.** Valores do tipo `boolean` podem ser comparados com $>$ e $<$.
**H.** Valores do tipo `int` podem estar entre $-2^{-31}$ e $2^{31}$.
**I.** O tipo `char` pode ser usado para representar pares de caracteres uma vez que variáveis deste tipo ocupam dois bytes na memória.
**J.** Os tipos de dados `double` e `long` não são equivalentes apesar de variáveis destes tipos ocuparem o mesmo espaço na memória.

**Exercício 2.10:** ⋆

Considerando a tabela 2.2, escolha o tipo de dado ou classe mais adequada para representar:

- O número de bytes em um disco rígido de computador moderno.
- O número de bytes em um disquete de computador.
- O tamanho em polegadas de um monitor de computador.
- O nome de um modelo de computador.
- O número de pontos em uma tela ajustada para a resolução $800 \times 600$.

**Exercício 2.11:** ⋆

Considerando a tabela 2.2, escolha o tipo de dado ou classe mais adequada para representar:

- O número de municípios de um estado do Brasil.
- O nome de um estado do Brasil.
- A população de um estado do Brasil.
- A área do Brasil em $km^2$.
- A população de todos os países do mundo.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.12:** ⋆

Considerando a tabela 2.2, escolha o tipo de dado ou classe mais adequada para representar:

- O resultado do arremesso de uma moeda.
- O resultado do arremesso de um dado.
- O naipe de uma carta de baralho.
- Um número sorteado na MegaSena.
- Um número sorteado na Loteria Federal.

**Exercício 2.13:** ⋆

Considerando a tabela 2.2, escolha o tipo de dado ou classe mais adequada para representar:

- A altura de uma pessoa em metros.
- O peso de uma pessoa em quilos.
- A temperatura corporal de uma pessoa.
- O sexo de uma pessoa.
- A altura de uma pessoa em milímetros.

**Exercício 2.14:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1 class IdentidadeDaPessoa
2   {
3   /**
4    * Declaração dos campos desta classe
5    */
6   int rg; // número do RG da pessoa
7   String nome; // nome do pai da pessoa
8   String nome; // nome da mãe da pessoa
9   } // fim da classe
```

**Exercício 2.15:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1 class Registro De Eleitor
2   {
3   /**
4    * Declaração dos campos desta classe
5    */
6   int tituloDeEleitor; // número do título do eleitor
7   String nome; // nome do eleitor
8   short zonaEleitoral; // número da zona eleitoral
9   } // fim da classe
```

**Exercício 2.16:** ⋆

Escreva a classe `Lampada` correspondente ao modelo da listagem 1.1. Que tipo de dado pode ser usado para representar o campo `estado` ?



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.17:** ⋆

Escreva na classe `Lampada` (veja o exercício 2.16) o método correspondente à resposta do exercício 1.7.

**Exercício 2.18:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Lampada
2    {
3    /**
4     * Declaração dos campos desta classe
5     */
6    private boolean ligada;
7    /**
8     * Este método acende a lâmpada
9     */
10   void acende a lâmpada()
11     {
12     ligada = 1;
13     }
14   /**
15    * Este método apaga a lâmpada
16    */
17   void apaga a lâmpada()
18     {
19     ligada = 0;
20     }
21   } // fim da classe
```

**Exercício 2.19:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class FaceDoDado
2    {
3    /**
4     * Declaração dos campos desta classe
5     */
6    int 1,2,3,4,5,6;
7    /**
8     * Declaração dos métodos desta classe
9     */
10   void 1()
11     {
12     System.out.println(1);
13     }
14   void 2()
15     {
16     System.out.println(2);
17     }
18   void 3()
19     {
20     System.out.println(3);
21     }
22   void 4()
23     {
24     System.out.println(4);
25     }
26   void 5()
27     {
28     System.out.println(5);
29     }
30   void 6()
31     {
32     System.out.println(6);
33     }
34   } // fim da classe
```

**Exercício 2.20:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DoisValores
2    {
3    /**
4     * Declaração dos campos desta classe
5     */
6    int valor1,valor2;
7    /**
8     * Declaração dos métodos desta classe
9     */
10   int maior()
11     {
12     if (valor1 > valor2)
13       return true;
14     else return false;
15     }
16   void menor()
17     {
18     if (valor1 < valor2)
19       return valor1;
20     else return valor2;
21     }
22   } // fim da classe
```

**Exercício 2.21:** ⋆

Modifique a resposta do exercício 2.16 para que a classe represente também o número de watts da lâmpada. Escreva um método `éEconômica` que retorne o valor booleano `true` se a lâmpada consumir menos de 40 watts e `false` caso contrário. *Dica:* a expressão `(a > b)` retorna `true` se `a` for maior do que `b` e `false` caso contrário.

**Exercício 2.22:** ⋆

Modifique o método `mostraData` da classe `Data` (listagem 2.7) para que o mês seja mostrado por extenso ao invés de numeral (isto é, quando o mês for 3, que o método imprima Março, etc.)

**Exercício 2.23:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class NumeroComplexo
2    {
3    /**
4     * Declaração dos campos desta classe
5     */
6    float real,imaginário;
7    /**
8     * Declaração dos métodos desta classe
9     */
10   float valor()
11     {
12     return real,imaginário;
13     }
14   } // fim da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.24:** ⋆

Escreva a classe `Ponto2D`, correspondente ao modelo da resposta do exercício 1.23.

**Exercício 2.25:** ⋆

Escreva a classe `Veiculo`, correspondente ao modelo da resposta do exercício 1.24.

**Exercício 2.26:** ⋆

Escreva a classe `VeiculoTerrestre`, correspondente ao modelo da resposta do exercício 1.25. Veja também o exercício 2.25.

**Exercício 2.27:** ⋆

A classe `Data` (listagem 2.7) pode representar datas não válidas, com os valores de dia, mês e ano iguais a zero. Modifique o método `mostraData` para que se a data encapsulada não for válida, uma mensagem "Data Inválida" seja impressa ao invés dos valores de dia, mês e ano (veja o exercício 1.13).

**Exercício 2.28:** ⋆

Escreva a classe `LampadaTresEstados` correspondente à resposta do exercício 1.5. Que tipo de dado pode ser usado para representar o campo `estado` ?

**Exercício 2.29:** ⋆

Escreva a classe `Lampada100Estados` correspondente à resposta do exercício 1.6. Considere também a resposta do exercício 2.28. Que tipo de dado pode ser usado para representar o campo `estado` ?

**Exercício 2.30:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Media
2    {
3    /**
4     * Declaração dos campos desta classe
5     */
6    double val1,val2,val3;
7    /**
8     * Declaração dos métodos desta classe
9     */
10   media2()
11     {
12     return (val1+val2)/2;
13     }
14   media3()
15     {
16     return (val1+val2+val3)/3;
17     }
18   } // fim da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.31:** ★

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Amplitude
2    {
3    /**
4     * Declaração dos campos desta classe
5     */
6    double val1,val2,val3;
7    /**
8     * Declaração dos métodos desta classe
9     */
10   double amplitude()
11     {
12     double amplitude2()
13       {
14       return val1-val2;
15       }
16     return amplitude2()-val3;
17     }
18   } // fim da classe
```

**Exercício 2.32:** ★

Escreva uma classe `HoraAproximada` que represente, em Java, o modelo do exercício 1.10.

**Exercício 2.33:** ★

Usando o exercício 2.32 como referência, escreva uma classe `HoraPrecisa` que represente, em Java, o modelo do exercício 1.11.

**Exercício 2.34:** ★

Escreva uma classe `Livro` que represente, em Java, o modelo do exercício 1.15.

**Exercício 2.35:** ★

Escreva uma classe `LivroLivraria` que represente, em Java, o modelo do exercício 1.16.

**Exercício 2.36:** ★

Escreva uma classe `LivroBiblioteca` que represente, em Java, o modelo do exercício 1.17.

**Exercício 2.37:** ★

Modifique o método `éIgual` da classe `Data` (listagem 2.7) para que uma data inválida seja considerada sempre diferente de qualquer outras.

**Exercício 2.38:** ★

O que aconteceria se todos os métodos da classe `Data` (listagem 2.7) fossem declarados com o modificador `private` ?

**Exercício 2.39:** ★

Escreva uma classe `Pais` que represente os dados gerais sobre um país (nome, área, população, etc.). Que campos e métodos esta classe deve ter ?

**Exercício 2.40:** ⋆

Escreva uma classe `AutomovelUsado` que represente os dados de um automóvel usado à venda, como ano, modelo, quilometragem rodada, combustível, preço pedido, etc. Que campos e métodos esta classe deve ter ?

**Exercício 2.41:** ⋆

Escreva uma classe `CadernoDeEnderecos` que represente os dados de uma pessoa, como nome, telefone, e-mail e endereço. Que campos e métodos esta classe deve ter ?

**Exercício 2.42:** ⋆

Escreva uma classe `Musica` que represente os dados de uma música, para uso em um banco de dados ou catálogo de músicas. Represente nesta classe o título, cantor e tempo de duração da música, usando para este último campo uma instância da classe `HoraPrecisa` (veja os exercícios 1.22 e 2.33).

**Exercício 2.43:** ⋆⋆

Escreva a classe `Contador` que encapsule um valor usado para contagem de itens ou eventos. Esta classe deve esconder o valor encapsulado de programadores-usuários, fazendo com que o acesso ao valor seja feito através de métodos que devem zerar, incrementar e imprimir o valor do contador.

**Exercício 2.44:** ⋆⋆

Modifique a classe `Lampada` para que esta contenha também um campo que indique quantas vezes a lâmpada foi acesa. Tente usar uma instância da classe `Contador` (veja o exercício 2.43). Em que método este campo deve ser modificado ?

**Exercício 2.45:** ⋆⋆

Escreva a classe `RegistroAcademico`, baseada na classe `RegistroAcademicoSimples` (listagem 2.2), fazendo com que todos os campos sejam privados e adicionando os métodos necessários ao funcionamento da classe.

**Exercício 2.46:** ⋆⋆

Considere os exercícios 2.44 e 2.43. Faça com que o contador que conta quantas vezes uma lâmpada foi acesa seja uma instância da classe `Contador`.

**Exercício 2.47:** ⋆⋆

Crie a classe `DataHora` que represente simultaneamente uma data e uma hora aproximada. *Dica:* o modelo pode conter instâncias das classes `Data` e `HoraAproximada`. Use os exercícios 1.12 e 2.32 como referência.

**Exercício 2.48:** ⋆⋆

Escreva a classe `Empregado`, correspondente à resposta do exercício 1.27.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.49:** ⋆⋆

Crie uma classe `Linha` para representar uma linha, unida por dois pontos no espaço cartesiano de duas dimensões, usando duas instâncias da classe `Ponto2D`, criada no exercício 2.24. Veja também o exercício 1.33.

**Exercício 2.50:** ⋆⋆

Crie uma classe `Retangulo` para representar um retângulo cujos pontos opostos sejam duas instâncias da classe `Ponto2D`, que deve ter sido criada no exercício 2.24. Veja também o exercício 1.34.

**Exercício 2.51:** ⋆⋆

Escreva a classe `PoligonoRegular`, correspondente à resposta do exercício 1.36.

**Exercício 2.52:** ⋆⋆

Escreva, na classe `Data`, um método `duplicaData` que receba, como argumento, uma outra instância da classe `Data` e duplique os valores dos campos da instância passada como argumento para os campos encapsulados.

**Exercício 2.53:** ⋆⋆⋆

Escreva uma classe `ContaBancariaSimplificada` que corresponda ao modelo na listagem 1.2. Considere que modificadores de acesso devam ser usados para os métodos e campos da classe.

**Exercício 2.54:** ⋆⋆⋆

Se os métodos `abreConta`, `deposita` e `retira` que devem ter sido criadas no exercício 2.53 forem criados como o modelo da listagem 1.2 sugere, alguns erros poderão ocorrer, como abrir uma conta com valor negativo, ou depositar ou retirar valores negativos. Modifique os métodos citados para que somente valores positivos sejam considerados pelos métodos.

**Exercício 2.55:** ⋆⋆⋆

Escreva a classe `LampadaQueimada`, correspondente à resposta do exercício 1.37.

**Exercício 2.56:** ⋆⋆⋆

Escreva uma classe `ModeloDeComputador` que encapsule valores que definam a configuração de um microcomputador (tipo de processador, memória RAM, tamanho do disco rígido, tamanho do monitor, por exemplo). Esta classe deve ter um método `calculaPreço` que calcule o preço do computador como sendo a soma do custo de seus componentes:

- Placa-mãe: R$800
- Opções de processadores: 600Mhz a R$700, 800Mhz a R$830, 933Mhz a R$910
- Opções de memória: 128, 256, 384 ou 512 Mb, cada 128Mb custa R$350.
- Opções de disco rígido: 20 Gb a R$300, 40 Gb a R$420, 60 Gb a R$500.
- Opções de monitor: 15 polegadas a R$320, 17 polegadas a R$520.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.57:** ⋆⋆⋆

Implemente a solução do exercício 1.38 na classe `RegistroAcademico`. Veja também o exercício 2.45.

**Exercício 2.58:** ⋆⋆⋆

Implemente a solução do exercício 1.39 na classe `RegistroAcademico`. Veja também o exercício 2.45.

**Exercício 2.59:** ⋆⋆⋆

Modifique a classe `RegistroAcademico` (veja o exercício 2.45), adicionando o campo `códigoDoCurso`. Modifique também o método que imprime os valores dos campos da classe para que este imprima o *nome* do curso ao invés do código. Invente vários códigos (números de dois dígitos) que representam diferentes cursos.

**Exercício 2.60:** ⋆⋆⋆

Modifique a classe `Retangulo` (exercício 2.50) para que esta contenha métodos para retornar a área e o perímetro do retângulo encapsulado. *Dica:* a classe `Ponto2D`, que tem duas instâncias como campos na classe `Retangulo`, deve ter seus campos privados e métodos que permitam o acesso aos valores dos campos.

**Exercício 2.61:** ⋆⋆⋆

Implemente a lógica correta de cálculo de anos bissextos e dias nos meses mostrada no exercício 1.42 na classe `Data`.

**Exercício 2.62:** ⋆⋆⋆

Uma das operações que podemos efetuar com datas é a comparação para ver se uma data ocorre antes de outra. O algoritmo para comparação é muito simples, e seus passos estão abaixo. Neste algoritmo, consideramos que `dia1`, `mês1` e `ano1` são os dados da primeira data, e que `dia2`, `mês2` e `ano2` são os dados da segunda data.

1. Se `ano1 < ano2` a primeira data vem antes da segunda.
2. Se `ano1 > ano2` a primeira data vem depois da segunda.
3. Se `ano1 == ano2` e `mês1 < mês2` a primeira data vem antes da segunda.
4. Se `ano1 == ano2` e `mês1 > mês2` a primeira data vem depois da segunda.
5. Se `ano1 == ano2` e `mês1 == mês2` e `dia1 < dia2` a primeira data vem antes da segunda.
6. Se `ano1 == ano2` e `mês1 == mês2` e `dia1 > dia2` a primeira data vem depois da segunda.
7. Se nenhum destes casos ocorrer, as datas são exatamente iguais.

Escreva um método `vemAntes` na classe `Data` (listagem 2.7) que receba como argumento outra instância da classe `Data` e implemente o algoritmo acima, retornando `true` se a data encapsulada vier antes da passada como argumento e `false` caso contrário. Se as datas forem exatamente iguais, o método deve retornar `true`.

**Exercício 2.63:** ★★★★

Escreva em Java uma classe `RestauranteCaseiro` que implemente o modelo descrito na figura 1.1 na seção 1.2 do capítulo 1. Para isto, crie também uma classe `MesaDeRestaurante` que represente uma mesa de restaurante conforme mostrado na figura 1.1. Algumas sugestões sobre a criação destas classes são:

- A classe `MesaDeRestaurante` deve ter campos para representar a quantidade de cada pedido feito, um método `adicionaAoPedido` que incrementa a quantidade de pedidos feitos, o método `zeraPedidos` que cancela todos os pedidos feitos, isto é, faz com que a quantidade de pedidos seja zero para cada item e o método `calculaTotal` que calcula o total a ser pago por aquela mesa.
- A classe `RestauranteCaseiro` deve ter várias campos que são instâncias da classe `MesaDeRestaurante`, para representar suas mesas.
- A classe `RestauranteCaseiro` também deve ter um método `adicionaAoPedido` que adicionará uma quantidade a um item de uma mesa. Este método deverá chamar o método `adicionaAoPedido` da mesa à qual o pedido está sendo adicionado.

**Exercício 2.64:** ★★★★

Escreva em Java a classe `NumeroComplexo` que represente um número complexo. A classe deverá ter os seguintes métodos:

- `inicializaNúmero`, que recebe dois valores como argumentos para inicializar os campos da classe
- `imprimeNúmero`, que deve imprimir o número complexo encapsulado usando a notação $a+bi$ onde $a$ é a parte real e $b$ a imaginária
- `éIgual`, que recebe outra instância da classe `NumeroComplexo` e retorna `true` se os valores dos campos encapsulados forem iguais aos da instância passada como argumento
- `soma`, que recebe outra instância da classe `NumeroComplexo` e soma este número complexo com o encapsulado usando a fórmula $(a+bi)+(c+di)=(a+c)+(b+d)i$
- `subtrai`, que recebe outra instância da classe `NumeroComplexo` e subtrai o argumento do número complexo encapsulado usando a fórmula $(a+bi)-(c+di)=(a-c)+(b-d)i$
- `multiplica`, que recebe outra instância da classe `NumeroComplexo` e multiplica este número complexo com o encapsulado usando a fórmula $(a+bi)*(c+di)=(ac-bd)+(ad+bc)i$
- `divide`, que recebe outra instância da classe `NumeroComplexo` e divide o número encapsulado pelo passado como argumento usando a fórmula $\frac{(a+bi)}{(c+di)}=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}i$



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 2.65:** ★★★★

Modifique a classe `Retangulo` (exercício 2.50) para que esta contenha dois métodos adicionais: um para verificar se uma instância da classe `Ponto2D` passada como argumento está localizada dentro da instância da classe `Retangulo`, que deverá retornar `true` se o ponto estiver dentro do retângulo, e outro para fazer o mesmo com uma instância da classe `Linha`. *Dica:* para verificar se um ponto está dentro do retângulo, verifique se as coordenadas do ponto estão dentro das coordenadas do retângulo. Considerando a figura 2.1, onde $(x1,y1)$ e $(x2,y2)$ são as coordenadas que definem o retângulo, o ponto $P1$ estaria fora do retângulo uma vez que a sua coordenada $y$ é menor do que a menor coordenada $y$ do retângulo. O ponto $P2$ estaria dentro do retângulo, e o ponto $P3$ também estaria fora do retângulo. Para verificar se uma linha está dentro ou fora do retângulo, basta verificar os dois pontos que formam suas extremidades: somente se os dois pontos estiverem dentro do retângulo a linha também estará: na figura 2.1, a linha $L2$ está dentro do retângulo, as linhas $L1$ e $L3$ não.
Veja também os exercícios 2.24, 2.49 e 2.60.

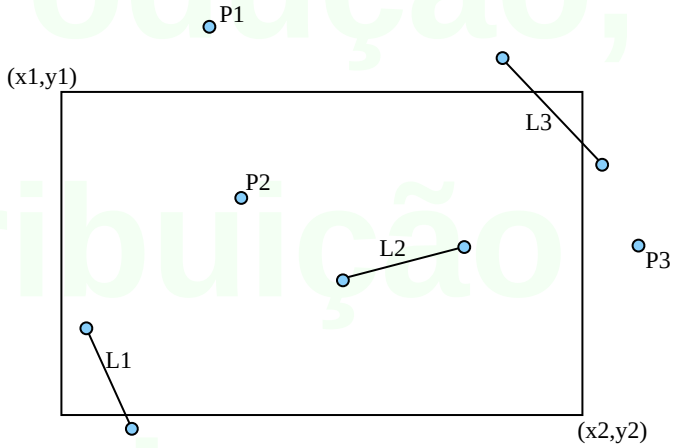



Figura 2.1: Pontos, linhas e retângulos no espaço cartesiano bidimensional.

# Capítulo 3

# Criando Aplicações em Java

## 3.1 Introdução

No capítulo 2 vimos que podemos criar classes que modelam entidades, pessoas, processos, etc. usando a linguagem de programação Java. Estas classes, além de conter campos necessários para representação dos dados inerentes aos modelos, podem conter métodos para processar estes dados.

*algoritmo*

Classes, no entanto, não são programas de computador - programas completos devem ser capazes de executar um *algoritmo*, ou seja, uma tarefa com passos definidos para a solução de um problema. O primeiro passo para a execução de um programa é chamado ponto de entrada. Os métodos das classes podem ser considerados algoritmos capazes de resolver problemas relacionados às classes a que pertencem, mas definitivamente não são soluções completas - não existe, claramente, um ponto inicial para o programa, a partir de onde as decisões e operaçõe serão feitas.

*ponto de entrada de um programa*

Neste capítulo veremos como podemos criar classes em Java que se comportam como aplicações ou programas, e os mecanismos de criação de instâncias de classes que representam modelos dentro destes programas.[1]

## 3.2 Criando aplicações em Java

*classe executável*

Em Java, podemos criar em uma classe um método especial[2] que será considerado o ponto de entrada de um programa. A presença deste método na classe fará com que a classe se torne executável, e dentro deste método poderemos ter a criação e manipulação de dados e instâncias de classes.

*método* `main`

Este método especial em Java é chamado `main`, similarmente à linguagens como C e C++. Para que a classe possa ser executada pela máquina virtual Java, não basta a existência de um método chamado `main` - este método deve ter obrigatoriamente os modificadores `public static` (nesta ordem), devendo retornar `void` e receber como argumento um array (agrupamento, conjunto ou lista) de instâncias da classe String, ou seja, deve ser declarado como `public static void main(String[] argumentos)`. Vale a pena lembrar que a declaração de métodos **não** deve ser

[1] Por enquanto aplicações com interfaces gráficas não serão criadas, somente aplicações que são executadas no modo texto dos sistemas operacionais, através de terminais. A criação de programas com interfaces gráficas será vista na parte III.

[2] Devemos sempre lembrar que não podemos ter métodos fora de classes em Java.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

terminada com ponto-e-vírgula.

Segue a explicação de cada um dos termos da declaração: na seção 2.6 vimos que o modificador `public` faz com que o método seja visível de qualquer outra classe, o que será uma exigência da máquina virtual Java para executá-lo. No capítulo 5 veremos o papel do modificador `static` - por enquanto fiquemos com a explicação simples de que um método declarado como `static` dispensa a criação de uma instância de sua classe para que possamos chamá-lo.

O "tipo de retorno" `void` indica que o método `main` não deverá retornar nada. O nome do método deve ser exatamente `main`, não podendo ter variações como maiúsculas, sublinhados, etc. O método deve receber como argumento um array de Strings, que deverá ter um nome (referência). O nome pode ser qualquer um que siga as regras de uso de nomes de campos em Java, mas o array de Strings deve ser declarado como `String[]`. Arrays serão discutidos com detalhes no capítulo 11. Este array de Strings conterá, para o método `main`, os argumentos passados para a classe executável via a linha de comando do terminal do computador. Informações sobre o uso dos dados da linha de comando serão vistas na seção 11.4.

Em princípio, o método `main` poderia pertencer a qualquer classe, até mesmo a uma classe que representa um modelo qualquer. Ao invés de usar este estilo de programação, vamos tentar fazer com que o método `main`, sempre que possível, seja o único método de uma classe, estando separado da representação dos modelos. Além de tornar os programas executáveis mais claros, esta prática serve para melhorar a distinção entre os diferentes papéis de programadores que usem linguagens orientadas a objetos, discutidos na seção 1.8.

Como primeiro exemplo, vamos ver uma outra classe que demonstra o uso do método `main` - algumas já foram vistas no capítulo anterior para demonstrar os modificadores de acesso. A classe `MaisDemoData` (listagem 3.1) demonstra[3] mais usos de instâncias da classe `Data` (listagem 2.7).

Listagem 3.1: A classe `MaisDemoData`, que demonstra mais usos de instâncias da classe `Data`.

```
1  /**
2   * A classe MaisDemoData, que demonstra mais usos da classe Data. Esta classe
3   * demonstra, em especial, o método main que permite a execução da classe, e o
4   * operador new que permite a criação de instâncias de classes.
5   */
6  class MaisDemoData // declaração da classe
7    {
8    /**
9     * O método main permite a execução desta classe. Este método contém declarações
10    * de algumas instâncias da classe Data, e demonstra seu uso.
11    * @param argumentos os argumentos que podem ser passados para o método via linha
12    *        de comando, mas que neste caso serão ignorados.
13    */
14   public static void main(String[] argumentos)
15     {
16     // Declaramos duas referências à instâncias da classe Data, mas não as
17     // inicializamos.
18     Data nascimentoDeGeorgesBizet;
19     Data nascimentoDePabloPicasso;
20     // Criamos três instâncias da classe Data. A instanciação é feita pelo operador
21     // new.
22     Data nascimentoDeJohannStrauss = new Data();
23     Data JohnSteinbeckGanhaPrêmioNobel = new Data();
24     Data nascimentoDeNancyCartwright = new Data();
```

[3] Classes que contém somente o método `main` e tem o nome iniciado por `Demo` serão consideradas, neste livro, programas demonstrativos de conceitos e classes.

```
25      // Declaramos três variáveis auxiliares para uso nas inicializações
26      byte d,m; short a;
27      // Inicializamos algumas das instâncias com um método da classe Data, usando
28      // as três variáveis auxiliares
29      d = 25; m = 10; a = 1838; // na verdade foi em 1825, o erro é proposital
30      nascimentoDeJohannStrauss.inicializaData(d,m,a);
31      d = 25; m = 10; a = 1962;
32      JohnSteinbeckGanhaPrêmioNobel.inicializaData(d,m,a);
33      // Podemos passar os argumentos diretamente para o método de inicialização se
34      // pudermos garantir que eles são dos tipos esperados
35      nascimentoDeNancyCartwright.inicializaData((byte)25,(byte)10,(short)1959);
36      // Até agora as referências nascimentoDeGeorgesBizet e nascimentoDePabloPicasso
37      // não foram inicializadas, mas podem ser usadas mesmo assim:
38      nascimentoDeGeorgesBizet = nascimentoDeJohannStrauss;
39      // Vamos ver se realmente funciona ?
40      nascimentoDeGeorgesBizet.mostraData(); // imprime 25/10/1838
41      // O ano do nascimento de Johann Strauss está errado ! Vamos corrigi-lo.
42      nascimentoDeJohannStrauss.inicializaData((byte)25,(byte)10,(short)1825);
43      // Imprimimos agora as duas datas, a que foi modificada...
44      nascimentoDeJohannStrauss.mostraData(); // imprime 25/10/1825
45      // e a que era uma cópia (duas referências para a mesma instância)
46      nascimentoDeGeorgesBizet.mostraData();  // imprime 25/10/1825 !!!
47      // Estas duas datas são iguais ?
48      System.out.println(JohnSteinbeckGanhaPrêmioNobel.
49                                 éIgual(nascimentoDeNancyCartwright)); // false
50      // Podemos comparar os anos das datas.
51      short ano1,ano2;
52      ano1 = JohnSteinbeckGanhaPrêmioNobel.retornaAno();
53      ano2 = nascimentoDeNancyCartwright.retornaAno();
54      if (ano1 > ano2)
55        {
56        System.out.println("John Steinbeck ganhou o prêmio Nobel de literatura depois");
57        System.out.println(" do nascimento de Nancy Cartwright.");
58        }
59      } // fim do método main
60
61    } // fim da classe MaisDemoData
62
63  // Datas obtidas em http://www.scopesys.com/today/
```

Alguns comentários sobre a listagem 3.1 são:

- A classe não contém campos, e só contém o método `main`, de acordo com a sugestão de estilo comentada acima. O método `main` declara várias instâncias da classe `Data`, que deve ter sido criada para uso neste programa.
- A classe que contém o método `main` é em geral muito mais simples ou concisa do que uma aplicação equivalente escrita em C ou outra linguagem procedural, pois parte das tarefas (no caso, inicialização com alguma verificação dos dados, impressão e comparação) é feita internamente pelos objetos.
- O método `main` é o ponto de entrada da aplicação ou programa, e os comandos dentro do método serão executados um a um, do início ao fim, exceto no caso de condicionais como os mostrados nas linhas 54 a 58, onde o bloco entre as linhas 55 e 58 só será executado se a condição na linha 54 for verdadeira.
- A palavra-chave `new` é usada para a criação de novas instâncias de classes, e é descrita na seção 3.3.
- Dentro do método `main` temos algumas variáveis declaradas (`d`, `m` e `a`) - devemos lembrar que estes não são considerados campos da classe pois não representam dados sobre um modelo, e sim valores auxiliares que serão usados por métodos.
- O compilador Java tenta, na medida do possível, adaptar valores constantes a tipos de dados, como mostrado nas atribuições à variáveis das linhas 29 e 31. Os valores numéricos

cast

constantes são considerados como sendo inteiros, mas convertidos para `byte`s e `short`s quando as atribuições são feitas. Infelizmente isto não ocorre com os argumentos para métodos: nos casos da linhas 35 e 42, os tipos dos dados devem ser explicitados (operação de *cast*) para que os métodos possam ser chamados corretamente.

## 3.3 A palavra-chave `new`

*new*

A criação e uso de instâncias de classes em métodos como o `main` mostrado na listagem 3.1 é feita através da palavra-chave `new`, como mostrado nas linhas 22 a 24. A instância criada com a palavra-chave `new` é associada a uma referência, desta forma a referência será usada para acessar a instância para que possamos fazer operações com ela (linhas 30, 32, 35 e outras).

*referência*

A classe na listagem 3.1 declara mais referências do que usa, e usa uma referência sem que uma nova instância tenha sido criada e associada explicitamente a ela (linha 38). Embora normalmente cada instância deva ter uma referência associada e vice-versa, é possível declarar referências que não são associadas a instâncias (mas que não poderão ser usadas antes da associação), múltiplas referências a uma mesma instância e até mesmo instâncias sem referências (instâncias anônimas), embora isto só seja útil quando tivermos que passar instâncias como argumentos para métodos.

A figura 3.1 mostra, graficamente, a situação das referências e instâncias criadas no programa da listagem 3.1.

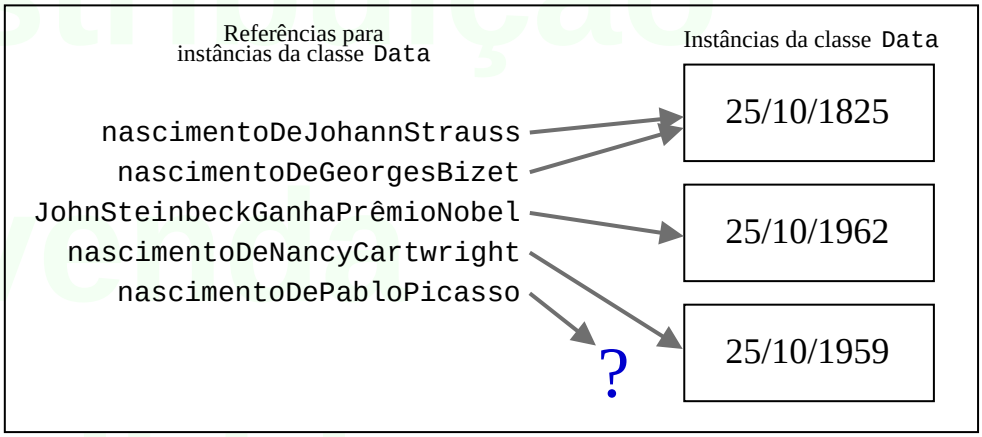



Figura 3.1: Referências e instâncias criadas na classe `MaisDemoData`

Alguns pontos que podem ser esclarecidos pela figura 3.1 e pela listagem 3.1 são:

*referências e instâncias*

- As referências *apontam* para as instâncias. No método `main` da classe `MaisDemoData`, `nascimentoDeJohannStrauss` e `nascimentoDeGeorgesBizet` apontam para a mesma instância. Uma instância da classe `Data` foi criada (na linha 23) e associada à referência `nascimentoDeJohannStrauss`, e na linha 39 a referência `nascimentoDeGeorgesBizet` foi feita igual à `nascimentoDeJohannStrauss`.

  *mais de uma referência para uma instância*

  Qualquer uma destas duas referências pode ser usada para executar métodos da instância. Podemos ver que os dados da instância foram modificados pela chamada ao método `inicializaData` pela referência `nascimentoDeJohannStrauss` na linha 43, e que o método `mostraData` foi chamado pela referência `nascimentoDeGeorgesBizet`, imprimindo a data modificada, provando que as duas referências acessam e/ou modificam a mesma instância.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

É importante notar que mesmo que duas instâncias de uma classe contenham os mesmos dados encapsulados, elas não são iguais, afinal ocupam posições diferentes na memória.

*referências sem instâncias associadas*

- A referência `nascimentoDePabloPicasso` foi declarada na linha 20, mas não teve nenhuma instância associada a ela, nem via a criação de instâncias com o `new` nem pela cópia de outra referência. Apesar da referência ser declarada de forma que possa ser associada a uma instância da classe `Data`, a associação nunca é feita (a referência não aponta para nenhuma instância), então a tentativa de chamar algum método da classe `Data` através da referência `nascimentoDePabloPicasso` causará um erro de compilação.
- Referências são independentes: se duas referências apontam para uma instância, e a primeira referência é modificada para apontar para outra instância, a segunda referência continua apontando para a primeira instância.

A criação de instâncias através do uso da palavra-chave `new` pode ocorrer dentro da própria classe cuja instância está sendo criada. Um exemplo disto será visto a seguir, com a classe `Ponto2D` que encapsula duas coordenadas que representam um ponto no espaço cartesiano.


Figura 3.2: Coordenadas no espaço cartesiano de duas dimensões

A figura 3.3 mostra alguns pontos no espaço cartesiano. Cada ponto terá dois valores associados a ele. A clase `Ponto2D` que representa estes pontos é mostrada na listagem 3.2.

Listagem 3.2: A classe `Ponto2D`, que encapsula um ponto no espaço cartesiano de duas dimensões.

```
1  /**
2   * A classe Ponto2D, que encapsula um ponto no espaço cartesiano de duas dimensões.
3   */
4  class Ponto2D // declaração da classe
5    {
6   /**
7    * Declaração dos campos da classe
8    */
9    private double x,y; // as coordenadas do ponto no espaço bidimensional
10
11  /**
12   * O método inicializaPonto2D recebe dois argumentos para inicializar os respectivos
13   * campos da classe Ponto2D. Não existem restrições aos valores que os dados podem
14   * receber, então a inicialização se limita a copiar os argumentos para os campos.
15   * @param _x o argumento correspondente à coordenada horizontal
16   * @param _y o argumento correspondente à coordenada vertical
17   */
18   public void inicializaPonto2D(double _x,double _y)
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
19      {
20      x = _x;
21      y = _y;
22      } // fim do método inicializaPonto2D
23
24    /**
25     * O método éIgual recebe uma instância da própria classe Ponto2D como argumento e
26     * verifica se as coordenadas encapsuladas na classe e na instância que foi passada
27     * são exatamente iguais, e retorna o valor booleano true se forem.
28     * @param outroPonto2D outra instância da própria classe Ponto2D
29     * @return true se as coordenadas encapsuladas forem exatamente iguais às
30     *         encapsuladas na instância passada como argumento, false caso contrário
31     */
32    public boolean éIgual(Ponto2D outroPonto2D)
33      {
34      if ((x == outroPonto2D.x) &&  // se as coordenadas horizontais forem iguais E
35          (y == outroPonto2D.y))    // se as coordenadas verticais forem iguais então
36        return true;      // o ponto é igual, retorna true
37      else
38        return false;     // o ponto é diferente, retorna false
39      } // fim do método éIgual
40
41    /**
42     * O método origem retorna uma nova instância da classe Ponto2D, que contém as
43     * coordenadas (0,0) (origem das coordenadas). Esta instância é criada por este
44     * método, e pode ser usada diretamente por aplicações ou classes que executem
45     * este método sem necessidade de usar a palavra-chave new.
46     * @return uma nova instância da classe Ponto2D representando a coordenada (0,0)
47     */
48    public Ponto2D origem()
49      {
50      Ponto2D temporário = new Ponto2D(); // a inicialização da instância é feita aqui
51      temporário.inicializaPonto2D(0,0);
52      return temporário; // retornamos a referência inicializada
53      } // fim do método origem
54
55    /**
56     * O método clona retorna uma nova instância da classe Ponto2D, que contém as
57     * mesmas coordenadas encapsuladas na classe, ou seja, quando este método for
58     * executado a partir de uma instância desta classe, ele retornará uma nova
59     * instância com os mesmos valores encapsulados.
60     * @return uma nova instância da classe Ponto2D contendo os valores encapsulados
61     */
62    public Ponto2D clona()
63      {
64      Ponto2D temporário = new Ponto2D(); // a inicialização da instância é feita aqui
65      temporário.inicializaPonto2D(x,y); // os valores encapsulados são usados
66      return temporário; // retornamos a referência inicializada
67      } // fim do método clona
68
69    /**
70     * O método toString é um método "mágico" que não recebe argumentos, e retorna uma
71     * String contendo os valores dos campos da classe formatados.
72     * @return uma String com os valores dos campos formatados como (x,y)
73     */
74    public String toString()
75      {
76      String resultado = "("+x+","+y+")";
77      return resultado;
78      } // fim do método toString
79
80    } // fim da classe Ponto2D
```

Os pontos de interesse da listagem 3.2 são:

- A classe `Ponto2D` tem alguns métodos que serão declarados para quase todas as classes que encapsulam dados: um método para inicializar os dados (`inicializaPonto2D`) e um

método para comparar os dados encapsulados com os de uma instância passada como argumento (éIgual).

- Dois dos métodos na classe Ponto2D criam novas instâncias da própria classe: origem e clona. O método origem cria uma nova instância de Ponto2D e a inicializa com os valores (0,0) - a origem do sistema cartesiano. O método é declarado como retornando uma referência para instância da classe Ponto2D, e retorna a referência temporário. O método clona é similar, mas retorna uma referência para nova instância da classe Ponto2D contendo os valores encapsulados.
- A classe Ponto2D tem um método especial: o método toString. Este método retorna os campos da classe formatados através de uma String. Este método deve ser declarado com o modificador de acesso public e retornar uma String. O uso deste método será mostrado na listagem 3.3.

*método* toString

Listagem 3.3: A classe DemoPonto2D, que demonstra usos da classe Ponto2D.

```
1  /**
2   * A classe DemoPonto2D, que demonstra usos da classe Ponto2D. Esta classe demonstra,
3   * em especial, onde é necessário ou não o uso do operador new para criar novas
4   * instâncias da classe Ponto2D.
5   */
6  class DemoPonto2D // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações
10   * de algumas instâncias da classe Ponto2D, e demonstra sua criação e uso.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Declaramos quatro referências a instâncias da classe Ponto2D, mas não
17     // criamos as instâncias associadas nem as inicializamos ainda.
18     Ponto2D p1;           // declaramos somente uma referência
19     Ponto2D p2,p3,p4;     // várias referências podem ser declaradas desta forma
20     // Vamos criar duas instâncias e associá-las às referências
21     p1 = new Ponto2D();
22     p2 = new Ponto2D();
23     // Vamos inicializar somente uma instância com valores
24     p1.inicializaPonto2D(-1.34,9.17);
25     // Como o método toString existe na classe Ponto2D, podemos imprimir diretamente
26     // as instâncias
27     System.out.println("As coordenadas de P1 são "+p1); // imprime (-1.34,9.17)
28     // Se a instância p2 não foi inicializada com valores, que valores ela contém ?
29     System.out.println("As coordenadas de P2 são "+p2); // imprime (0.0,0.0)
30     // Podemos criar instâncias sem usar diretamente o new !
31     p3 = p1.clona();   // o método clona retorna uma nova instância
32     p4 = p1.origem();  // o método origem retorna uma nova instância
33     System.out.println("As coordenadas de P3 são "+p3); // imprime (-1.34,9.17)
34     System.out.println("As coordenadas de P4 são "+p4); // imprime (0.0,0.0)
35     // Que pontos são iguais aos outros ?
36     System.out.println("P1 está na mesma posição de P2 ? "+p1.éIgual(p2)); // false
37     System.out.println("P1 está na mesma posição de P3 ? "+p1.éIgual(p3)); // true
38     System.out.println("P1 está na mesma posição de P4 ? "+p1.éIgual(p4)); // false
39     System.out.println("P2 está na mesma posição de P4 ? "+p2.éIgual(p4)); // true
40     // Podemos ter instâncias sem referências !
41     System.out.println(new Ponto2D()); // imprime (0.0,0.0)
42     }  // fim do método main
43
44   } // fim da classe DemoPonto2D
```

Os pontos de interesse da listagem 3.3 são:

- O método `toString` da classe `Ponto2D` (listagem 3.2) permite que o conteúdo da instância da classe possa ser formatado como uma String. Este método tem um nome "mágico" - se este método existir em uma classe qualquer, as instâncias destas classes poderão ser impressas e/ou processadas como uma String diretamente, sem que o método precise ser chamado explicitamente. Na listagem 3.3, os conteúdos das instâncias `p1` e `p2` são impressos diretamente, e o método `toString` é chamado implicitamente.
- Os métodos `clona` e `origem` (listagem 3.2) criam novas instâncias da classe `Ponto2D` e as retornam, então podemos atribuir o resultado destes métodos diretamente à referências sem precisar inicializar estas. Isto é feito com as referências `p3` e `p4`, que recebem novas instâncias de `Ponto2D`.

*instâncias sem referências associadas*

- Na listagem 3.1 foi demonstrado que a relação entre referências e instâncias não é necessariamente de uma para uma: podemos ter várias referências apontando para a mesma instância. Na linha 42 da listagem 3.3 temos o caso oposto: uma instância é criada através da palavra-chave `new` mas não é associada à nenhuma referência. Esta instância é passada como argumento para o método `System.out.println`, que automaticamente chamará o método `toString`, que retornará uma String com os campos da instância formatados, que serão impressos. Vemos então que é possível ter instâncias que não sejam associadas à referências, mas elas terão utilidade limitada.

## 3.4 A palavra-chave `null`

`null`

Vimos na listagem 3.1 que referências podem apontar para instâncias que devem ter sido criadas através da palavra-chave `null`, ou podem não apontar para nanhum lugar. Existe uma terceira possibilidade: referências podem ser iguais a `null`, indicando que elas não foram inicializadas mas tem algum "valor" associado à elas, apesar deste "valor" não poder ser usado como uma instância poderia.

A diferença pode parecer sutil, mas o compilador aceita que existam referências que não apontam para nada durante a compilação contanto que nenhum método da instância que deveria ser associada à referência seja chamado. Se em algum ponto do código houver uma tentativa de chamada de método para uma referência que não seja inicializada, o compilador não permitirá a compilação do programa.

Se inicializarmos as referências com `null`, o compilador permitirá a compilação do programa, mas se algum método da instância que deveria ser associada à referência for executado, um erro de execução ocorrerá. Desta forma, podemos inicializar as referências com `null` caso haja necessidade de fazer alguma inicialização condicional, que só poderá ser determinada em tempo de execução do programa.

A inicialização de referências com `null` é exemplificada na classe `DemoReferencias`, na listagem 3.4. Note que a classe não pode ser compilada por causa de erros propositais.

Listagem 3.4: A classe `DemoReferencias`, que demonstra mais exemplos de referências.

```
1 /**
2  * A classe DemoReferencias, que demonstra mais exemplos de referências e sua
3  * associação a instâncias da classe Data. Esta classe também demonstra o uso da
4  * palavra-chave null para denotar referências que não apontam para nenhuma
5  * instância.
6  * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
7  */
```

```
8  class DemoReferencias // declaração da classe
9    {
10   /**
11    * O método main permite a execução desta classe. Este método contém declarações
12    * de algumas instâncias da classe Data e suas referências.
13    * @param argumentos os argumentos que podem ser passados para o método via linha
14    *        de comando, mas que neste caso serão ignorados.
15    */
16   public static void main(String[] argumentos)
17     {
18     // Declaramos três referências a instâncias da classe Data, e inicializamos duas
19     // com o "valor" null
20     Data nascimentoDeEdgarDegas = null;                    // 19/7/1834
21     Data nascimentoDeHerbertMarcuse;                       // 19/7/1898
22     Data jogosOlímpicosEmHelsinki = null;                  // 19/7/1952
23     // Criamos uma nova instância da classe Data e a associamos à uma das referências
24     nascimentoDeEdgarDegas = new Data();
25     // Usamos a referência para inicializar os campos da instância
26     nascimentoDeEdgarDegas.inicializaData((byte)30,(byte)1,(short)1948);
27     nascimentoDeEdgarDegas.mostraData();
28     // Dereferenciamos a referência - ela volta a apontar para null
29     nascimentoDeEdgarDegas = null;
30     // Se o programa pudesse ser compilado, a linha abaixo causaria um erro em tempo de
31     // execução que interromperia a execução do programa. A próxima linha causaria o
32     // mesmo erro, mas nunca será alcançada.
33     nascimentoDeEdgarDegas.mostraData();
34     jogosOlímpicosEmHelsinki.mostraData();
35     // Uma inicialização condicional: a instância nascimentoDeHerbertMarcuse só
36     // será inicializada se true for igual a false, ou seja, nunca.
37     if (true == false)
38       {
39       nascimentoDeHerbertMarcuse = new Data();
40       nascimentoDeHerbertMarcuse.inicializaData((byte)21,(byte)6,(short)1965);
41       }
42     // A linha abaixo causa um erro de compilação, já que a referência
43     // nascimentoDeHerbertMarcuse pode não ter sido inicializada no bloco acima. O
44     // compilador não tem como saber se o bloco será executado, mesmo nos casos mais
45     // óbvios.
46     nascimentoDeHerbertMarcuse.mostraData();
47     } // fim do método main
48
49 } // fim da classe DemoReferencias
```

Os pontos de interesse na listagem 3.4 são:

- O compilador não indicará erro na linha 33, onde a referência `nascimentoDeEdgarDegas` executa o método `mostraData`, pois a referência foi inicializada (mesmo que tenha sido com `null`), mas o programa não será executado corretamente porque em tempo de execução a máquina virtual Java verá que não existe uma instância associada à esta referência.
- O compilador mostrará mensagens de erro e não terminará a compilação da classe mostrada na listagem 3.4 pois na linha 47 há uma tentativa de chamar o método `mostraData` usando a instância `nascimentoDeHerbertMarcuse` que pode não ter sido inicializada (o compilador não tem como saber se o bloco que inicializa a instância e seus dados será executado). Se a instância `nascimentoDeHerbertMarcuse` fosse inicializada com `null`, o programa poderia ser compilado (mesmo que erros de execução ocorressem).
- É possível *dereferenciar* uma referência, ou seja, fazer com que ela aponte para `null` ou até mesmo para outra instância, bastando igualar a referência à `null`, a outra referência ou a outra instância criada através da palavra-chave `new`, como mostrado na linha 30. Se a referência já criada for igualada à `null`, esta não poderá ser usada para acessar dados ou chamar métodos da instância a qual estava associada anteriormente. O compilador Java não permite que igualemos uma referência à outra referência que não tenha sido inicializada (isto é, que somente tenha sido declarada).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

## 3.5 Escrevendo classes e aplicações mais interativas

Uma característica das classes e aplicações mostradas até agora é que todos os dados que são usados para demonstrar as classes são codificados nas aplicações - em outras palavras, em todas as vezes que as aplicações forem executadas, o resultado será o mesmo.

Para aumentar a interatividade (e mesmo a variedade e aplicabilidade das classes e aplicações), os valores processados podem ser lidos do teclado: neste caso, quando um valor for lido do teclado, a aplicação interromperá o processamento e aguardará até que o usuário entre um valor e pressione a tecla ENTER.

*A classe* `Keyboard`

A classe `Keyboard`, que é descrita com detalhes no apêndice B, permite a leitura de valores de tipos nativos e instâncias da classe String, que podem ser armazenados em variáveis e campos do programa. Para que a classe `Keyboard` possa ser usada, ela deve estar presente no mesmo diretório onde estão as suas classes, ou em um diretório especificado pela linha de comando do interpretador Java. A solução mais simples é copiar a classe `Keyboard` para o diretório onde as classes e aplicações estão sendo desenvolvidas. O apêndice B.1 mostra como é possível referenciar classes em outros diretórios.

A listagem 3.3 demonstra o uso de métodos da classe `Keyboard`. Para mais exemplos e informações, assim como para ver o código-fonte da classe `Keyboard`, veja o apêndice B.

Listagem 3.5: A classe `DemoPonto2DK`, que demonstra usos da classe `Ponto2D`, que serão inicializados com valores lidos do teclado.

```
1  /**
2   * A classe DemoPonto2DK, que demonstra usos da classe Ponto2D, que serão
3   * inicializados com valores lidos do teclado.
4   */
5  class DemoPonto2DK // declaração da classe
6    {
7    /**
8     * O método main permite a execução desta classe. Este método contém declarações
9     * de algumas instâncias da classe Ponto2D, e demonstra sua criação e uso. Os
10    * valores para inicialização dos pontos serão lidos do teclado.
11    * @param argumentos os argumentos que podem ser passados para o método via linha
12    *        de comando, mas que neste caso serão ignorados.
13    */
14   public static void main(String[] argumentos)
15     {
16     // Declaramos duas referências a instâncias da classe Ponto2D, e criamos as
17     // instâncias associadas.
18     Ponto2D p1 = new Ponto2D();
19     Ponto2D p2 = new Ponto2D();
20     // Vamos ler dois valores do tipo double do teclado para inicialização
21     System.out.print("Entre um valor de ponto flutuante (coordenada x):");
22     double x = Keyboard.readDouble();
23     System.out.print("Entre outro valor de ponto flutuante (coordenada y):");
24     double y = Keyboard.readDouble();
25     p1.inicializaPonto2D(x,y);
26     // Vamos ler diretamente dois valores do tipo double (sem usar variáveis)
27     System.out.print("Entre dois valores de ponto flutuante, pressionando ENTER ");
28     System.out.print("após entrar cada um:");
29     // Inicializamos a instância p2 com os valores lidos diretamente.
30     p2.inicializaPonto2D(Keyboard.readDouble(),Keyboard.readDouble());
31     // Como o método toString existe na classe Ponto2D, podemos imprimir diretamente
32     // as instâncias
33     System.out.println("As coordenadas de P1 são "+p1); // imprime P1
34     System.out.println("As coordenadas de P2 são "+p2); // imprime P2
35     }  // fim do método main
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
36
37  } // fim da classe DemoPonto2DK
```

A classe `Keyboard` somente contém métodos estáticos, ou seja, a classe não precisa ser instanciada através de uma referência e da palavra-chave `new`. Métodos estáticos serão vistos no capítulo 5.

O uso mais comum dos métodos da classe `Keyboard` é demonstrado nas linhas 23 e 25 - o resultado dos métodos de leituras é colocado em variáveis ou campos. Na linha 31, mostramos que o resultado de um método (`readDouble`) pode ser passado diretamente como argumento para outro método (`inicializaPonto`), pois os argumentos esperados pelo método `inicializaPonto` são iguais ou compatíveis com o retornado por `readDouble`.

## 3.6 Exercícios do capítulo 3

**Exercício 3.1:** ⋆

Explique, com suas palavras, porque uma classe como a `Ponto2D` (listagem 3.2) não pode ser executada.

**Exercício 3.2:** ⋆

Escreva um programa em Java que imprima o seu nome.

**Exercício 3.3:** ⋆

Escreva um programa em Java que leia o seu nome do teclado e imprima-o com uma mensagem qualquer. Veja o apêndice B para exemplos.

**Exercício 3.4:** ⋆

Escreva um programa em Java que use várias instâncias da classe `Lampada` (veja o exercício 2.16).

**Exercício 3.5:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo. Se não houverem erros, explique o que a classe faz.

```
1  class CalculaPiAproximado
2    {
3    public static double main(String[] args)
4      {
5      double piAproximado = 355./113.;
6      return piAproximado;
7      }
8    } // fim da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 3.6:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoImpressao
2    {
3    main(String[] args)
4      {
5      System.out.println("7+2="+(7+2));
6      System.out.println("7-2="+(7-2));
7      System.out.println("7*2="+(7*2));
8      System.out.println("7/2="+(7/2));
9      return true;
10     }
11   } // fim da classe
```

**Exercício 3.7:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Demonstração de Divisões
2    {
3    public static void main()
4      {
5      double n = 120;
6      double d = 17;
7      System.out.println(n+" dividido por "+d+" é "+(n/d));
8      System.out.println("o resto da divisão é "+(n%d));
9      }
10   } // fim da classe
```

**Exercício 3.8:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  public static void main(String[] args)
2    {
3    Data2 hoje = new Data2();
4    hoje.inicializaData(7,1,2001);
5    hoje.imprimeData();
6    }
7
```

**Exercício 3.9:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class AloMundo
2    public static double main(String[] args)
3      System.out.println("Alô Mundo !");
```

**Exercício 3.10:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DatasIguais
2    {
3    public void static main(String[] args)
4      {
5      Data2 hoje = new Data2();
6      hoje.inicializaData(7,1,2001);
7      Data2 amanha;
8      amanha.inicializaData(8,1,2001);
9      System.out.println(amanha.éIgual(hoje));
10     }
11   } // fim da classe
```

**Exercício 3.11:** ⋆

Corrija, compile e execute o programa na listagem 3.4.

**Exercício 3.12:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Atribuicoes
2    {
3    public static void main(String[] args)
4      {
5      Data a;
6      Data b = new Data();
7      b = null;
8      b = a;
9      }
10   } // fim da classe
```

**Exercício 3.13:** ⋆

Escreva uma aplicação em Java que demonstre o uso de várias instâncias da classe `Pais`, que deve ter sido criada como resposta ao exercício 2.39.

**Exercício 3.14:** ⋆

Escreva uma aplicação em Java que demonstre o uso de instâncias da classe `Contador`, que deve ter sido criada como resposta ao exercício 2.43.

**Exercício 3.15:** ⋆

Escreva uma aplicação em Java que demonstre o uso de instâncias da classe `PoligonoRegular`, que deve ter sido criada como resposta ao exercício 2.51.

**Exercício 3.16:** ⋆⋆

Uma instância da classe `Ponto2D` foi criada na linha 42 da listagem 3.3. Os dados encapsulados nesta instância podem ser modificados ? Porquê ?

**Exercício 3.17:** ⋆⋆

Usando as classes `Veiculo` e `VeiculoTerrestre`, que devem ter sido criadas como resposta aos exercícios 2.25 e 2.26, escreva uma aplicação em Java que declare várias instâncias das duas classes. Em particular, tente descrever o mesmo veículo usando duas instâncias, uma da classe `Veiculo` e outra da classe `VeiculoTerrestre`.

**Exercício 3.18:** ⋆⋆

Escreva uma aplicação em Java que demonstre o uso de instâncias das classes `Livro`, `LivroLivraria` e `LivroBiblioteca` (veja os exercícios 2.34, 2.35 e 2.36).

**Exercício 3.19:** ⋆⋆

Escreva uma aplicação em Java que demonstre o uso de instâncias da classe `Lampada` que incorpore um contador de quantas vezes foi acesa (veja o exercício 2.44).

**Exercício 3.20:** ⋆⋆

Escreva uma aplicação em Java que demonstre o uso de instâncias da classe `DataHora` (veja o exercício 2.47).

**Exercício 3.21:** ⋆⋆

Escreva uma aplicação que demonstre o uso de instâncias da classe `ContaBancariaSimplificada` que deve ter sido criada como resposta ao exercício 2.53. Demonstre como a transferência de valores de uma instância da classe para outra pode ser feita através de chamadas aos métodos `deposita` e `retira`. Tente fazer com que os dados que serão usados nas classes sejam lidos do teclado (veja o apêndice B).

**Exercício 3.22:** ⋆⋆⋆

Demonstre o uso de instâncias da classe `RegistroAcademico`, que deve ter sido escrita com todas as modificações sugeridas nos exercícios do capítulo 3. Veja os exercícios 2.57, 2.58 e 2.59.

**Exercício 3.23:** ⋆⋆⋆

A classe abaixo pode ser compilada sem erros. Quando for executado, o programa imprimirá que o resultado da comparação na linha 11 é `true` mas o resultado da comparação na linha 12 é `false`. Explique porquê.

```
1  class DemoDataCopiada
2    {
3    public static void main(String[] argumentos)
4      {
5      Data lançamentoDaAtlantis18 = new Data();
6      Data inícioDeOperaçãoDoHAL  = new Data();
7      Data morteDeCharlesHuggins;
8      lançamentoDaAtlantis18.inicializaData((byte)12,(byte)1,(short)1997);
9      inícioDeOperaçãoDoHAL.inicializaData ((byte)12,(byte)1,(short)1997);
10     morteDeCharlesHuggins = lançamentoDaAtlantis18;
11     System.out.println(lançamentoDaAtlantis18 == morteDeCharlesHuggins);
12     System.out.println(lançamentoDaAtlantis18 == inícioDeOperaçãoDoHAL);
13     }
14   }
```

**Exercício 3.24:** ⋆⋆⋆

Escreva uma aplicação que demonstre o uso de instâncias da classe `NumeroComplexo` que deve ter sido criada como resposta ao exercício 2.64. Demonstre o uso de todas as operações.

## 3.7 Exercícios complementares do capítulo 3

**Exercício 3.25:** ⋆

O que acontecerá se tentamos imprimir uma instância de uma classe que não tem o método `toString` ? Demonstre com um pequeno programa.

**Exercício 3.26:** ⋆

Escreva o método `toString` na classe `Data`, de forma que o mês seja mostrado por extenso ao invés de numeral (veja o exercício 2.22).

**Exercício 3.27:** ⋆

Escreva o método `clona` na classe `Data`, que retorne uma nova instância que é uma cópia da própria data.

**Exercício 3.28:** ⋆

Escreva o método `toString` para as classes `Livro`, `LivroLivraria` e `LivroBiblioteca` (exercícios 2.34, 2.35 e 2.36).

**Exercício 3.29:** ⋆

Escreva o método `toString` para a classe `ContaBancariaSimplificada` (exercício 2.53).

**Exercício 3.30:** ⋆

Retire as partes que explicitam a conversão de dados nas linhas 35 e 42 da listagem 3.1. O que acontece ? Porquê ?

**Exercício 3.31:** ⋆

Crie o método `criaRevertido` para a classe `Ponto2D` (listagem 3.2) que retorne uma nova instância da classe onde os valores encapsulados `x` e `y` são revertidos.

**Exercício 3.32:** ⋆⋆

Escreva o método `toString` para a classe `Retangulo` (exercício 2.50), reaproveitando o método `toString` da classe `Ponto2D` (listagem 3.2).

**Exercício 3.33:** ⋆⋆

Escreva o método `toString` para a classe `ModeloDeComputador` (exercício 2.56).

**Exercício 3.34:** ⋆⋆

Crie o método `temEixoComum` para a classe `Ponto2D` (listagem 3.2) que receba uma outra instância da classe `Ponto2D` e retorne o valor booleano `true` se as coordenadas horizontais e/ou verticais encapsuladas forem iguais às da instância passada como argumento. Por exemplo, as coordenadas (1,2) e (1,-17) tem eixo comum, as (-9,0) e (-9,0) também tem, mas as (13,-8) e (8,-22) não tem eixo comum.

**Exercício 3.35:** ⋆⋆

Crie o método `distância` para a classe `Ponto2D` (listagem 3.2) que recebe uma outra instância da classe `Ponto2D` e retorna um valor do tipo `double` correspondente à distância Euclidiana entre o `Ponto2D` encapsulado e o passado como argumento. *Dica:* a distância Euclidiana `d` entre um ponto com coordenadas $(x_1, y_1)$ e outro ponto com coordenadas $(x_2, y_2)$ é calculada por $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, que pode ser escrita em Java como `d = Math.sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))`, onde `Math.sqrt` é o método em Java que calcula a raiz quadrada.

**Exercício 3.36:** ⋆⋆

Crie o método `distânciaDaOrigem` para a classe `Ponto2D` (listagem 3.2) que não recebe nenhum argumento, mas calcula a distância Euclidiana entre as coordenadas encapsuladas e a origem do sistema de coordenadas. Para isto, dentro do método, crie uma instância de `Ponto2D` correspondente à origem e passe-a como argumento para o método `distância`, que deve ter sido escrito como resposta ao exercício 3.35.

**Exercício 3.37:** ⋆⋆⋆

Crie o método `éPróximo` para a classe `Ponto2D` (listagem 3.2) que recebe uma outra instância da classe `Ponto2D` e um limiar (valor do tipo `double`) como argumentos, calculando a distância entre as coordenadas encapsulados e as coordenadas da instância passada como argumento, retornando o valor booleano `true` se a distância for menor do que o limiar passado como argumento. Por exemplo, se o ponto encapsulado vale (3,3), o passado como argumento vale (4,1) e o limiar for 3.0 o método deve retornar `true` já que a distância entre os dois pontos (2.236) é menor do que o limiar. Se o limiar fosse 2.0 o método deveria retornar `false`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

# Capítulo 4

# Construtores e Sobrecarga

## 4.1 Introdução

No capítulo 3 vimos que podemos criar aplicações que utilizam instâncias de classes definidas pelo usuário ou já existentes. Para criar instâncias, precisamos usar a palavra-chave `new` que criará a instância da classe, e, em geral, associará a instância recém-criada à uma referência, para que métodos da instância possam ser executados.

*inicialização de campos de instâncias*

Até agora, além de inicializar a instância da classe com a palavra-chave `new`, usamos métodos para inicializar os campos da instâncias. O uso destes métodos é de responsabilidade do programador usuário das classes - para criar as instâncias o compilador obriga o uso da palavra-chave `new`, mas não obriga o uso dos métodos de inicialização. Desta maneira, por esquecimento, um programador usuário pode criar a instância de uma classe mas não inicializar os seus dados.

Para exemplificar os erros que podem ocorrer quando instâncias são inicializadas mas os dados encapsulados não o são, consideremos as classes `RegistroAcademicoSemConstrutor` (listagem 4.1) que encapsula os dados de um registro acadêmico simples (basicamente para cálculo da mensalidade) e a classe `DemoRegistroAcademicoSemConstrutor` (listagem 4.2) que demonstra usos de instâncias desta classe.

Listagem 4.1: A classe `RegistroAcademicoSemConstrutor`, que encapsula alguns dados de um registro acadêmico.

```
1  /**
2   * A classe RegistroAcademicoSemConstrutor, que contém campos para representar
3   * dados simples sobre um registro acadêmico.
4   */
5  class RegistroAcademicoSemConstrutor // declaração da classe
6    {
7   /**
8    * Declaração dos campos da classe
9    */
10   private String nomeDoAluno; // O nome do aluno
11   private int númeroDeMatrícula; // O número de matrícula
12   private byte códigoDoCurso; // O código do curso do aluno (1..4)
13   private double percentualDeCobrança; // O percentual a ser cobrado do aluno, 0 a 100%
14
15  /**
16   * O método inicializaRegistroAcademicoSemConstrutor recebe argumentos para
17   * inicializar os campos da classe RegistroAcademicoSemConstrutor.
18   * @param n o nome do aluno
19   * @param m o número de matrícula
20   * @param c o código do curso
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
21    * @param p o percentual de bolsa
22    */
23   public void inicializaRegistroAcademicoSemConstrutor(String n,int m,byte c,double p)
24     {
25     nomeDoAluno = n; númeroDeMatrícula = m;
26     códigoDoCurso = c; percentualDeCobrança = p;
27     } // fim do método inicializaRegistroAcademicoSemConstrutor
28
29  /**
30   * O método calculaMensalidade calcula e retorna a mensalidade do aluno usando
31   * o código do seu curso e o percentual de cobrança.
32   * @return o valor da mensalidade do aluno
33   */
34   public double calculaMensalidade()
35     {
36     double mensalidade = 0; // valor deve ser inicializado
37     // Primeiro, dependendo do curso do aluno, determina a mensalidade básica
38     if (códigoDoCurso == 1) // Arquitetura
39       mensalidade = 450.00;
40     if (códigoDoCurso == 2) // Ciência da Computação
41       mensalidade = 500.00;
42     if (códigoDoCurso == 3) // Engenharia da Computação
43       mensalidade = 550.00;
44     if (códigoDoCurso == 4) // Zootecnia
45       mensalidade = 380.00;
46     // Agora calcula o desconto com o percentual de cobrança. Se o percentual de
47     // cobrança for zero, a mensalidade também o será.
48     if (percentualDeCobrança == 0) mensalidade = 0;
49     // Senão, calculamos com uma fórmula simples.
50     else mensalidade = mensalidade * 100.0 / percentualDeCobrança;
51     return mensalidade;
52     } // fim do método calculaMensalidade
53
54   } // fim da classe RegistroAcademicoSemConstrutor
```

Listagem 4.2: A classe `DemoRegistroAcademicoSemConstrutor`, que demonstra o uso de instâncias da classe `RegistroAcademicoSemConstrutor`.

```
1  /**
2   * A classe DemoRegistroAcademicoSemConstrutor, que demonstra o uso de instâncias da
3   * classe RegistroAcademicoSemConstrutor.
4   */
5  class DemoRegistroAcademicoSemConstrutor // declaração da classe
6    {
7    /**
8     * O método main permite a execução desta classe. Este método contém declarações de
9     * algumas instâncias da classe RegistroAcademicoSemConstrutor, e demonstra seu uso.
10    * @param argumentos os argumentos que podem ser passados para o método via linha
11    *        de comando, mas que neste caso serão ignorados.
12    */
13   public static void main(String[] argumentos)
14     {
15     // Declaramos duas referências a instâncias da classe
16     // RegistroAcademicoSemConstrutor, e as inicializamos com a palavra-chave new
17     RegistroAcademicoSemConstrutor michael = new RegistroAcademicoSemConstrutor();
18     RegistroAcademicoSemConstrutor roberto = new RegistroAcademicoSemConstrutor();
19     // Chamamos o método de inicialização somente para uma das instâncias
20     michael.inicializaRegistroAcademicoSemConstrutor("Michael Goodrich",34980030,
21                                                      (byte)2,100.0);
22     // Calculamos a mensalidade dos dois alunos - mesmo que um não tenha sido
23     // inicializado !
24     System.out.println("A mensalidade do Michael é "+michael.calculaMensalidade());
25     System.out.println("A mensalidade do Roberto é "+roberto.calculaMensalidade());
26     } // fim do método main
27
28   } // fim da classe DemoRegistroAcademicoSemConstrutor
```

Podemos ver que na listagem 4.2 uma instância da classe `RegistroAcademicoSemConstrutor` não teve seus dados inicializados, o que não impede o compilador de compilar corretamente a classe, nem a máquina virtual Java de a interpretar. O resultado do programa na listagem 4.2 é mostrado a seguir:

```
A mensalidade do Michael é 500.0
A mensalidade do Roberto é NaN
```

`NaN` (Not a Number)

A mensalidade do primeiro aluno foi calculada corretamente: a mensalidade de seu curso é 500 reais e o percentual de cobrança é 100% (sem bolsa). No caso do segundo aluno, os dados simplesmente não foram informados - para a sua instância os valores numéricos foram considerados como zero, e a mensalidade calculada como $0 * 100.0/0.0$ (veja a linha 47 da listagem 4.1). A divisão de zero por zero (quando ambos são valores de ponto flutuante) é representada por Java como o valor `NaN` (*Not a Number*, não é um número).

Em muitas situações será necessário forçar o programador usuário a passar dados para as instâncias criadas em classes e programas para que estas tenham sentido. Isto pode ser feito usando *construtores*. Este capítulo discutirá a criação e uso de construtores e de sobrecarga de métodos, que permite que criemos métodos com nomes iguais mas funções diferentes.

## 4.2 O que são construtores ?

*construtores*

*Construtores* são métodos especiais, que são chamados automaticamente quando instâncias são criadas através da palavra-chave `new`. Através da criação de construtores, podemos garantir que o código que eles contém será executado antes de qualquer outro código em outros métodos, já que uma instância de uma classe só pode ser usada depois de ter sido criada com `new`, o que causará a execução do construtor.

*valores* default

Construtores são particularmente úteis para inicializar campos de instâncias de classes para garantir que, quando métodos destas instâncias forem chamados, elas contenham valores específicos e não os *default*. Caso os campos de uma instância não sejam inicializados, os seguintes valores serão adotados:

- Campos do tipo `boolean` são inicializados automaticamente com a constante `false`.
- Campos do tipo `char` são inicializados com o caracter cujo código Unicode é zero e que é impresso como um espaço.
- Campos de tipos inteiros (`byte`, `short`, `long`, `int`) ou de ponto flutuante (`float`, `double`) são automaticamente inicializados com o valor zero, do tipo do campo declarado.
- Instâncias de qualquer classe, inclusive da classe `String`, são inicializadas automaticamente com `null`.

As diferenças básicas entre construtores e outros métodos são:

*regras para criação de construtores*

- Construtores devem ter **exatamente** o mesmo nome da classe a que pertencem, inclusive considerando maiúsculas e minúsculas.
- Construtores não podem retornar nenhum valor, nem mesmo `void`, portanto devem ser declarados sem tipo de retorno.
- Construtores não devem receber modificadores como `public` ou `private`, e serão públicos se a classe for pública. Não há muito sentido em declarar um construtor como sendo `private`, a não ser quando um construtor privado for chamado de outro construtor público. Veremos na seção 4.3 que é possível termos mais de um método ou construtor com o mesmo nome.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

*executando construtores*

A razão pela qual os construtores tem regras mais fixas para nomenclatura do que métodos é que quando uma instância de uma classe que tem construtores for inicializada com a palavra-chave `new`, o compilador executará automaticamente o construtor, precisando então saber exatamente qual é o nome deste. Uma outra diferença significativa entre construtores e métodos comuns é que o programador não pode chamar construtores diretamente - somente quando a instância for inicializada com `new`.

Para exemplificar o papel e importância de construtores em uma classe, consideremos a classe `EventoAcademico`, na listagem 4.3, que representa um evento acadêmico como congresso, simpósio ou reunião.

Listagem 4.3: A classe `EventoAcademico`, que representa um evento acadêmico.

```
1  /**
2   * A classe EventoAcademico representa um evento acadêmico, como um congresso ou
3   * encontro, que se realiza em determinado período entre datas, local, e com um certo
4   * número de participantes.
5   */
6  class EventoAcademico // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10   */
11   private String nomeDoEvento,localDoEvento;
12   private Data inícioDoEvento,fimDoEvento;
13   private int númeroDeParticipantes;
14
15  /**
16   * O construtor para a classe EventoAcademico, que recebe argumentos para
17   * inicializar os campos da classe.
18   * @param i o início do evento
19   * @param f o início do evento
20   * @param n o nome do evento
21   * @param l o local do evento
22   * @param num o número de participantes do evento
23   */
24   EventoAcademico(String n,String l,Data i,Data f,int num)
25     {
26     nomeDoEvento = n;  localDoEvento = l;
27     // É necessário inicializar as instâncias internas da classe Data, e não somente
28     // igualá-las aos valores passados, caso contrário elas simplesmente serão outras
29     // referências para instâncias da classe Data na classe que chamou o construtor.
30     inícioDoEvento = new Data();
31     inícioDoEvento.inicializaData(i.retornaDia(),i.retornaMês(),i.retornaAno());
32     fimDoEvento = new Data();
33     fimDoEvento.inicializaData(f.retornaDia(),f.retornaMês(),f.retornaAno());
34     númeroDeParticipantes =  num;
35     } // fim do construtor
36
37   /**
38    * O método toString não recebe argumentos, e retorna uma String contendo os valores
39    * dos campos da classe formatados.
40    * @return uma String com os valores dos campos formatados como um relatório simples
41    */
42   public String toString()
43     {
44     String relatório = ""; // no início do método a String é vazia
45     relatório = relatório + "Evento: "+nomeDoEvento+"\n";
46     relatório = relatório + "Local:  "+localDoEvento+"\n";
47     relatório = relatório + "De:     "+inícioDoEvento.retornaDia()+"/"+
48                                       inícioDoEvento.retornaMês()+"/"+
49                                       inícioDoEvento.retornaAno()+" a "+
50                                       fimDoEvento.retornaDia()+"/"+
51                                       fimDoEvento.retornaMês()+"/"+
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
52                                      fimDoEvento.retornaAno()+"\n";
53     relatório = relatório + "Participantes: "+númeroDeParticipantes+"\n";
54     return relatório;
55     } // fim do método toString
56
57   } // fim da classe EventoAcademico
```

Alguns pontos interessantes da classe `EventoAcademico` são:

- O construtor da classe, declarado na linha 25, é bem similar a um método para inicialização dos dados encapsulados na classe, mas segue as regras de nomenclatura de construtores.
- Quando houverem instâncias de outras classes sendo usadas como campos de uma classe, estas instâncias podem também ser inicializadas no construtor. Desta maneira, o construtor vai criando e inicializando todos os campos necessários.
- Na classe `EventoAcademico` existem duas referências a instâncias da classe `Data` que **devem** ser inicializadas explicitamente, ao invés de simplesmente copiadas. A inicialização é feita através da palavra-chave `new`, e o método `inicializaData` da classe `Data` é chamado, usando como argumentos os valores obtidos da instância passada como argumento para o construtor. Desta forma teremos uma cópia exata dos dados da data passada como argumento, e não uma referência apontando para a mesma instância (veja a listagem 3.1 e a figura 3.1 para mais detalhes sobre cópias de referências).

Para exemplificar o uso de instâncias da classe `EventoAcademico`, usamos a aplicação na classe `DemoEventoAcademico`, mostrada na listagem 4.4.

Listagem 4.4: A classe `DemoEventoAcademico`, que demonstra o uso de instâncias da classe `EventoAcademico`.

```
1  /**
2   * A classe DemoEventoAcademico, que demonstra o uso de instâncias da classe
3   * EventoAcademico.
4   */
5  class DemoEventoAcademico // declaração da classe
6    {
7    /**
8     * O método main permite a execução desta classe. Este método contém declarações de
9     * algumas instâncias da classe EventoAcademico, e demonstra seu uso. Algumas
10    * instâncias da classe Data deverão ser criadas e passadas como argumento para o
11    * construtor da classe EventoAcademico.
12    * @param argumentos os argumentos que podem ser passados para o método via linha
13    *        de comando, mas que neste caso serão ignorados.
14    */
15   public static void main(String[] argumentos)
16     {
17     // Declaramos algumas referências a instâncias da classe EventoAcademico
18     // mas ainda não as inicializamos
19     EventoAcademico SBED1998;
20     EventoAcademico ISER1999;
21     // Algumas instâncias da classe Data serão úteis para inicializar as instâncias
22     // da classe EventoAcademico.
23     Data data1 = new Data();
24     Data data2 = new Data();
25     // Inicializamos as datas e a instância SBED1998
26     data1.inicializaData((byte)20,(byte)4,(short)1998);
27     data2.inicializaData((byte)27,(byte)4,(short)1998);
28     SBED1998 = new EventoAcademico("Simpósio Brasileiro de Educação à Distância",
29                                    "Londrina (PR)",
30                                    data1,data2,940);
31     // Reaproveitando as datas, inicializamos a instância ISER1999
32     data1.inicializaData((byte)28,(byte)11,(short)1999);
33     data2.inicializaData((byte)4,(byte)12,(short)1999);
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
34      ISER1999 = new EventoAcademico("International Symposium on Educational Robotics",
35                                     "Taipei, Taiwan",
36                                     data1,data2,1308);
37      // Imprimimos os dados das instâncias através de chamadas implícitas ao método
38      // toString da classe EventoAcademico
39      System.out.println(SBED1998);
40      System.out.println(ISER1999);
41      } // fim do método main
42
43  } // fim da classe DemoEventoAcademico
```

- Instâncias da classe `EventoAcademico` só podem ser inicializadas com a palavra-chave `new` se argumentos forem passados para seu construtor. Desta forma, para esta classe, não é possível criar instâncias que tenham valores não-inicializados.
- Duas instâncias temporárias da classe `Data` são criadas na classe `DemoEventoAcademico` (linhas 24 e 25) somente para que sejam passadas como argumentos para os construtores da classe `EventoAcademico`. Estas instâncias são reaproveitadas ou reusadas com valores diferentes, mas não existem problemas de cópias de referências uma vez que, internamente na classe `EventoAcademico`, novas instâncias internas são criadas. Em resumo, mesmo depois que os conteúdos das instâncias `data1` e `data2` são modificados (linhas 33 e 34 da listagem 4.4) as datas representadas internamente na instância `SBED1998` da classe `EventoAcademico` não são modificadas.
- Podemos ver a praticidade de se ter um método `toString` na classe: podemos pedir ao método `System.out.println` que imprima a instância da classe `EventoAcademico`, e `System.out.println` executará automaticamente o método `toString`, imprimindo a String resultante.

*construtor* default

Mesmo quando as classes criadas pelo programador não tem um construtor declarado explicitamente, o compilador Java cria um construtor *default*, que não recebe argumentos nem executa nenhum código. Quando o programador de classes cria um ou mais construtores, o compilador não inclui o construtor default.

## 4.3 Sobrecarga de métodos

Em algumas ocasiões, será útil ou interessante poder executar um método em uma classe passando mais ou menos argumentos, conforme a necessidade. Por exemplo, consideremos a classe `ContaBancaria` que modela uma conta bancária, e seu construtor, que recebe dados para inicialização dos membros da classe, como nome, saldo inicial, se a conta é especial ou não, etc.

Consideremos que na maioria dos casos, contas serão abertas com saldo zerado e não serão contas especiais - se o construtor exige que os argumentos sejam passados, teremos que especificá-los todas as vezes que formos criar instâncias para esta classe. Seria mais prático ter *duas* versões do construtor, uma para a qual passaríamos somente os dados relevantes para inicialização da instância, sendo que os outros seriam considerados valores *default*, e outra versão para a qual passaríamos todos os dados.

Uma abordagem ingênua para a solução deste problema seria ter métodos diferenciados para inicialização, por exemplo, `abreContaBancária` que seria declarado esperando que valores para todos os campos fossem passados e `abreContaBancáriaComAlgunsDosDados` que esperaria somente alguns dos campos, e preencheria os outros com valores *default*. O problema com esta abordagem é que deve ficar a cargo do programador usuário chamar um dos métodos, e erros

podem ocorrer, em especial porque existirão vários métodos que fazem essencialmente a mesma função (preencher campos de instâncias ou inicializá-las) com nomes diferentes. Evidentemente a idéia de ter vários métodos com nomes diferentes não funcionaria de qualquer jeito para construtores, que devem ter o nome exatamente igual ao da classe.

*assinaturas de métodos*

Java (e outras linguagens orientadas a objetos) permitem a criação de métodos com nomes iguais, contanto que as suas *assinaturas* sejam diferentes. A assinatura de um método é composta de seu nome mais os tipos de argumentos que são passados para este método, independente dos nomes de variáveis usadas na declaração do método. Por exemplo, a assinatura do construtor declarado na linha 25 da listagem 4.3 é `EventoAcademico(String,String,Data,Data,int)`.

O tipo de retorno do método **não** é considerado como sendo parte da assinatura: não podemos ter dois métodos com o mesmo nome, tipo de argumentos mas tipo de retorno diferente[1]. Como é possível ter dois métodos com o mesmo nome contanto que as suas assinaturas sejam diferentes, seria possível criar dois construtores para a classe `EventoAcademico`, um que recebe todos os dados necessários para a inicialização dos campos e outro que só recebe os dados essenciais, inicializando os outros campos com valores *default*.

*sobrecarga de métodos*

A possibilidade de criar mais de um método com o mesmo nome e assinaturas diferentes é conhecida como *Sobrecarga de métodos*. A decisão sobre qual método será chamado quando existem dois ou mais métodos será feita pelo compilador, baseado na assinatura dos métodos.

Para dar um exemplo mais detalhado de construtores e métodos sobrecarregados, consideremos a classe `RoboSimples0`, na listagem 4.5, que modela um pequeno robô móvel de maneira bastante simplificada. Este robô é identificado por um nome, pode estar orientado em uma direção cardeal (Norte, Sul, Este ou Oeste), é capaz de se movimentar no espaço bidimensional na direção para a qual está orientado, e pode relatar a sua posição e direção atuais.

Listagem 4.5: A classe `RoboSimples0`, que encapsula dados de um robô móvel simulado.

```
1  /**
2   * A classe RoboSimples0 representa um robô que tem uma posição qualquer no espaço
3   * de duas dimensões, pode modificar a posição e direção (se movimentar, somente para
4   * a frente) e informar a sua posição e direção. Este robô é um modelo
5   * supersimplificado de um pequeno robô móvel. O nome da classe tem um zero pois esta
6   * classe ainda será modificada.
7   */
8  class RoboSimples0 // declaração da classe
9    {
10  /**
11   * Declaração dos campos da classe
12   */
13   private String nomeDoRobô;
14   private int posiçãoXAtual,posiçãoYAtual;
15   private char direçãoAtual;
16
17  /**
18   * O construtor para a classe RoboSimples0, que recebe argumentos para inicializar
19   * todos os campos da classe.
20   * @param n o nome do robô
21   * @param px a posição X atual
22   * @param py a posição Y atual
23   * @param d a direção atual
```

[1] É possível e em alguns casos até útil ter métodos com o mesmo nome e número de argumentos mas que retornem tipos diferentes. Considere um método que retorne o maior de dois números: um poderia ser declarado como `public int maior(int a,int b)` e outro como `public double maior(double a,double b)`

```
24    */
25    RoboSimples0(String n,int px,int py,char d)
26      {
27      nomeDoRobô = n;
28      posiçãoXAtual = px;
29      posiçãoYAtual = py;
30      direçãoAtual = d;
31      } // fim do construtor com todos os argumentos
32
33   /**
34    * O construtor para a classe RoboSimples0, que recebe somente o nome do robô e assume
35    * que este robô está na posição (0,0) e direção norte.
36    * @param n o nome do robô
37    */
38    RoboSimples0(String n)
39      {
40      nomeDoRobô = n;
41      posiçãoXAtual = 0;
42      posiçãoYAtual = 0;
43      direçãoAtual = 'N';
44      } // fim do construtor com somente o nome do robô como argumento
45
46   /**
47    * O construtor para a classe RoboSimples0, que não recebe argumentos e assume que o
48    * robô não tem nome (String vazia), e que este robô está na posição (0,0) e direção
49    * norte.
50    */
51    RoboSimples0()
52      {
53      nomeDoRobô = "";
54      posiçãoXAtual = 0;
55      posiçãoYAtual = 0;
56      direçãoAtual = 'N';
57      } // fim do construtor com somente o nome do robô como argumento
58
59   /**
60    * Esta versão do método move modifica a posição do robô em uma unidade na direção
61    * em que o robô está. Este método não recebe argumentos.
62    */
63    public void move()
64      {
65      if (direçãoAtual == 'N') posiçãoYAtual = posiçãoYAtual+1;
66      if (direçãoAtual == 'S') posiçãoYAtual = posiçãoYAtual-1;
67      if (direçãoAtual == 'E') posiçãoXAtual = posiçãoXAtual+1;
68      if (direçãoAtual == 'O') posiçãoXAtual = posiçãoXAtual-1;
69      } // fim do método move
70
71   /**
72    * Esta versão do método move modifica a posição do robô em um número de unidades na
73    * direção em que o robô está.
74    * @param passos o número de "passos" para o robô
75    */
76    public void move(int passos)
77      {
78      if (direçãoAtual == 'N') posiçãoYAtual = posiçãoYAtual+passos;
79      if (direçãoAtual == 'S') posiçãoYAtual = posiçãoYAtual-passos;
80      if (direçãoAtual == 'E') posiçãoXAtual = posiçãoXAtual+passos;
81      if (direçãoAtual == 'O') posiçãoXAtual = posiçãoXAtual-passos;
82      } // fim do método move
83
84   /**
85    * O método mudaDireção permite que a direção do robô seja mudada depois dele ter
86    * sido criado.
87    * @param novaDireção a nova direção para o robô.
88    */
89    public void mudaDireção(char novaDireção)
90      {
91      direçãoAtual = novaDireção;
92      } // fim do método mudaDireção
```

```
93
94   /**
95    * O método toString não recebe argumentos, e retorna uma String contendo os valores
96    * dos campos da classe formatados.
97    * @return uma String com os valores dos campos formatados.
98    */
99    public String toString()
100     {
101     String posição = "Nome do robô:"+nomeDoRobô+"\n";
102     posição = posição+"Posição do robô: ("+posiçãoXAtual+","+
103                                           posiçãoYAtual+")\n";
104     posição = posição+"Direção do robô:"+direçãoAtual;
105     return posição;
106     } // fim do método toString
107
108   } // fim da classe RoboSimples0
```

Alguns pontos de interesse da listagem 4.5 são:

- Existem três construtores para a classe, declarados nas linhas 26, 39 e 52, cada um recebendo um determinado número de argumentos para a inicialização das instâncias. O corpo dos três construtores é bem similar, a única real diferença sendo que quando um argumento não é passado, ele é inicializado explicitamente com um valor *default*.
- Como foram declarados construtores para esta classe, o compilador não incluiu automaticamente o construtor *default*, sem argumentos, mas um construtor sem argumentos (mas com funções bem definidas) foi escrito pelo programador da classe.
- Existem também métodos comuns sobrecarregados na listagem 4.5: o método `move`, que movimenta o robô tem duas versões, uma sem argumentos (declarada na linha 64) que considera que o robô dará um único "passo" na direção para a qual está orientado; e uma versão com argumentos (declarada na linha 77) que são o número de "passos" a serem dados.

A classe `DemoRoboSimples0`, mostrada na listagem 4.6, demonstra o uso de instâncias da classe `RoboSimples0`, em especial chamadas à construtores e métodos sobrecarregados.

Listagem 4.6: A classe `DemoRoboSimples0`, que demonstra a criação e uso de instâncias da classe `RoboSimples0`.

```
1  /**
2   * A classe DemoRoboSimples0, que demonstra o uso de instâncias da classe
3   * RoboSimples0.
4   */
5  class DemoRoboSimples0 // declaração da classe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * algumas instâncias da classe RoboSimples0, e demonstra seu uso. Basicamente alguns
10   * robôs simulados serão criados e movidos.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Declaramos algumas referências a instâncias da classe RoboSimples0
17     // e as inicializamos com construtores diversos.
18     RoboSimples0 groucho = new RoboSimples0("Groucho",20,10,'S');
19     RoboSimples0 chico = new RoboSimples0("Chico");
20     RoboSimples0 semNome = new RoboSimples0();
21     // Movimentamos estes robôs
22     groucho.move(10);
23     chico.mudaDireção('E');
24     chico.move(5);
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
25      semNome.move();
26      // Onde os robôs estão agora ?
27      System.out.println(groucho);
28      System.out.println(chico);
29      System.out.println(semNome);
30      } // fim do método main
31
32  } // fim da classe DemoRoboSimples0
```

Os pontos de interesse da listagem 4.6 são:

- A inicialização através do construtor e uso de alguns métodos da classe `RoboSimples0` ficou mais prática com os métodos sobrecarregados: somente os argumentos necessários são passados para os construtores e métodos, como descrito a seguir.
- A instância `groucho` será inicializada e o construtor com todos os argumentos (declarado na linha 26 da listagem 4.5, será chamado, já que a sua assinatura é igual à chamada pela palavra-chave `new`. Similarmente, a instância `chico` chamará o construtor declarado na linha 39 da listagem 4.5, e a instância `semNome` será inicializada com o construtor sem argumentos, declarado na linha 52 da listagem 4.5.
- As instâncias podem ter o método `move` chamado com argumentos (linhas 23 e 25 da listagem 4.6), ou sem argumentos (linha 26) - o compilador decicirá qual dos métodos deve ser chamado para cada caso.

### 4.3.1 A palavra-chave `this`

Revisitando a classe `RoboSimples`, podemos ver que parte do código é repetido entre os construtores e métodos sobrecarregados, o que é de se esperar, uma vez que a função destes é similar. Como o código é repetido, podemos esperar que a sua manutenção seja trabalhosa, pois se parte do código ou algoritmo tiver que ser mudado terá que ser mudado em vários pontos da classe. Suponha, por exemplo, que o robô possa se mover também na diagonal: neste caso boa parte do conteúdo dos métodos `move` deverá ser reescrita.

A classe `RoboSimples` pode ser reescrita de forma que a redundância entre trechos de código seja minimizada. Isto pode ser feito se pudermos executar uns métodos a partir de outros. Por exemplo, o método que move o robô sem receber argumentos é funcionalmente igual ao método que recebe argumentos, se passarmos o valor 1 como argumento, bastando chamar o método `move` com argumentos de dentro do método `move` sem argumentos - o compilador não se confundirá pois sabe qual método deverá ser chamado baseado nas assinaturas.

`this`

Para os construtores, a tarefa é mais complicada: não podemos chamar um construtor diretamente. Felizmente Java cria, internamente para cada instância, uma “auto-referência”, ou seja, uma referência à própria instância. Esta referência é representada pela palavra-chave `this`. Para chamar um construtor de dentro de outro, basta usar a palavra-chave `this` substituindo o nome do construtor[2]. Construtores não podem ser chamados indiscriminadamente de dentro de qualquer método: existem algumas regras para chamada de construtores que são:

*executando construtores a partir de outros*

1. Somente construtores podem chamar construtores como subrotinas.
2. Se um construtor for chamado a partir de outro, a chamada deve ser a primeira linha de código dentro do corpo do construtor.
3. Construtores não são chamados pelos seus nomes, e sim por `this`.

[2]Métodos podem ser chamados de dentro de outros métodos com a sintaxe `this.nomeDoMétodo`, mas a palavra-chave `this` é completamente opcional neste caso.

4. Construtores podem chamar outros métodos. Por exemplo, pode ser interessante ter um construtor e um método que inicializem as instâncias, e chamar o método de dentro do construtor. Métodos **não** podem chamar construtores, nem mesmo com `this`.
5. Construtores não podem ser chamados recursivamente: um construtor só pode chamar diretamente outro construtor, e não a si próprio.

A classe `RoboSimples0` foi reescrita e batizada `RoboSimples`, de forma que seus métodos chamem uns aos outros quando aplicável. O resultado é mostrado na listagem 4.7.

Listagem 4.7: A classe `RoboSimples`, que encapsula dados de um robô móvel simulado (com melhorias).

```
1  /**
2   * A classe RoboSimples representa um robô que tem uma posição qualquer no espaço
3   * de duas dimensões, pode modificar a posição e direção (se movimentar, somente para
4   * a frente) e informar a sua posição e direção. Este robô é um modelo
5   * supersimplificado de um pequeno robô móvel. Esta classe apresenta melhorias e
6   * simplificações em relação à classe RoboSimples0.
7   */
8  class RoboSimples // declaração da classe
9    {
10  /**
11   * Declaração dos campos da classe
12   */
13   private String nomeDoRobô;
14   private int posiçãoXAtual,posiçãoYAtual;
15   private char direçãoAtual;
16
17  /**
18   * O construtor "completo" para a classe RoboSimples, que recebe argumentos para
19   * inicializar todos os campos da classe.
20   * @param n o nome do robô
21   * @param px a posição X atual
22   * @param py a posição Y atual
23   * @param d a direção atual
24   */
25   RoboSimples(String n,int px,int py,char d)
26     {
27     nomeDoRobô = n;
28     posiçãoXAtual = px;
29     posiçãoYAtual = py;
30     direçãoAtual = d;
31     } // fim do construtor com todos os argumentos
32
33  /**
34   * O construtor para a classe RoboSimples, que recebe somente o nome do robô e assume
35   * que este robô está na posição (0,0) e direção norte.
36   * @param n o nome do robô
37   */
38   RoboSimples(String n)
39     {
40     // Chama o construtor completo passando a posição e direção como constantes.
41     this(n,0,0,'N');
42     } // fim do construtor com somente o nome do robô como argumento
43
44  /**
45   * O construtor para a classe RoboSimples, que não recebe argumentos e assume que o
46   * robô não tem nome (String vazia), e que este robô está na posição (0,0) e direção
47   * norte.
48   */
49   RoboSimples()
50     {
51     // Chama o construtor completo passando o nome como uma String vazia e a posição
52     // e direção como constantes.
53     this("",0,0,'N'); // Poderíamos escrever também this("") !
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
54      } // fim do construtor com somente o nome do robô como argumento
55
56    /**
57     * Esta versão do método move modifica a posição do robô em uma unidade na direção
58     * em que o robô está. Este método não recebe argumentos.
59     */
60     public void move()
61       {
62       move(1);
63       } // fim do método move
64
65    /**
66     * Esta versão do método move modifica a posição do robô em um número de unidades na
67     * direção em que o robô está.
68     * @param passos o número de "passos" para o robô
69     */
70     public void move(int passos)
71       {
72       if (direçãoAtual == 'N') posiçãoYAtual = posiçãoYAtual+passos;
73       if (direçãoAtual == 'S') posiçãoYAtual = posiçãoYAtual-passos;
74       if (direçãoAtual == 'E') posiçãoXAtual = posiçãoXAtual+passos;
75       if (direçãoAtual == 'O') posiçãoXAtual = posiçãoXAtual-passos;
76       } // fim do método move
77
78    /**
79     * O método mudaDireção permite que a direção do robô seja mudada depois dele ter
80     * sido criado.
81     * @param novaDireção a nova direção para o robô.
82     */
83     public void mudaDireção(char novaDireção)
84       {
85       direçãoAtual = novaDireção;
86       } // fim do método mudaDireção
87
88    /**
89     * O método toString não recebe argumentos, e retorna uma String contendo os valores
90     * dos campos da classe formatados.
91     * @return uma String com os valores dos campos formatados.
92     */
93     public String toString()
94       {
95       String posição = "Nome do robô:"+nomeDoRobô+"\n";
96       posição = posição+"Posição do robô: ("+posiçãoXAtual+","+
97                                             posiçãoYAtual+")\n";
98       posição = posição+"Direção do robô:"+direçãoAtual;
99       return posição;
100      } // fim do método toString
101
102   } // fim da classe RoboSimples
```

Alguns pontos de interesse na listagem 4.7 são:

*construtor "completo"*

- O primeiro construtor (declarado na linha 27) é apelidado "completo" pois inicializa explicitamente todos os campos da classe com argumentos passados. O segundo construtor (declarado na linha 40) recebe apenas o nome, e delega ao primeiro construtor a inicialização, passando para este a posição $(0,0)$ e a direção `'N'` como constantes. O terceiro construtor (declarado na linha 51) faz a mesma coisa, passando também o nome do robô como uma constante (String vazia).

*executando métodos em cascata*

- Construtores e métodos podem ser chamados em cascata: o corpo do terceiro construtor poderia ser escrito como `this();`, chamando então o segundo construtor, que por sua vez chamaria o primeiro.
- O método `move`, sem argumentos, também se beneficia da chamada de outros métodos: ele simplesmente delega o cálculo da nova posição ao outro método `move`, chamando este com o argumento 1.

Em resumo, o uso da sobrecarga de métodos em Java facilita as tarefas do programador usuário de classes, que chama o método adequado aos dados que quer passar. Ao programador das classes cabe definir que métodos e que argumentos estarão disponíveis para o programador usuário, dando flexibilidade às classes mas tentando, ao mesmo tempo, evitar a proliferação de métodos sobrecarregados sem necessidade.

### 4.3.2 Cuidados com sobrecarga de métodos

*promoção de tipos de dados*

Um cuidado adicional deve ser tomado pelo programador das classes quando for criar métodos sobrecarregados: Java permite que alguns tipos nativos de dados sejam *promovidos*, isto é, aceitos como sendo de outros tipos contanto que nada se perca na representação. Desta forma, um valor do tipo `byte` pode ser aceito por um método que espere um valor do tipo `int`, já que este pode representar bytes sem perda de informação.

*rebaixamento de tipos de dados*
cast

O mesmo não ocorre em casos onde um método ou construtor espere um certo tipo de dados e a chamada ao método contenha, como argumento, um tipo de dados que nem sempre pode ser contido no esperado: por exemplo, nem sempre um valor do tipo `double` pode ser representado por um valor do tipo `int`. Em muitos casos, podemos usar a conversão explícita ou *cast* para forçar o rebaixamento, geralmente com perda de precisão.

Quando for necessário determinar qual construtor ou método sobrecarregado deve ser chamado, o compilador pode, se for necessário, promover tipos de dados para que um construtor ou método com a assinatura adequada seja usado, mas nunca fará o rebaixamento, o que pode causar erros de compilação como os mostrados na listagem 4.8.

Listagem 4.8: A classe `DemoRoboSimplesComErros`, que demonstra o uso de chamadas com argumentos incorretos à construtores e métodos da classe `RoboSimples`.

```
1  /**
2   * A classe DemoRoboSimplesComErros, que demonstra o uso de instâncias da classe
3   * RoboSimples, com alguns erros causados pelos argumentos aos construtores e métodos
4   * sobrecarregados.
5   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
6   */
7  class DemoRoboSimplesComErros // declaração da classe
8    {
9   /**
10   * O método main permite a execução desta classe. Este método contém declarações de
11   * algumas instâncias da classe RoboSimples, mas argumentos de tipos diferentes são
12   * passados para os construtores e métodos.
13   * @param argumentos os argumentos que podem ser passados para o método via linha
14   *        de comando, mas que neste caso serão ignorados.
15   */
16   public static void main(String[] argumentos)
17     {
18     // Declaramos algumas referências a instâncias da classe RoboSimples e as
19     // inicializamos com construtores com argumentos diversos, mas diferentes dos
20     // esperados pelos construtores.
21     RoboSimples número5 = new RoboSimples("Número 5",'N','5',(char)69); // OK
22     byte coordX1 = -3;
23     byte coordY1 = 8;
24     RoboSimples gort = new RoboSimples("Gort",coordX1,coordY1,'E'); // OK
25     float coordX2 = 10.0f;
26     float coordY2 = 22.0f;
27     RoboSimples rosie = new RoboSimples("Rosie",coordX2,coordY2,'S'); // Erro
28     RoboSimples data = new RoboSimples(true); // Erro
29     RoboSimples clonado = new RoboSimples(número5); // Erro
30     // Chamamos métodos para estes robôs
31     byte distância = 12;
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
32      rosie.move(distância);
33      número5.move(3.2); // Erro
34      data.move(false); // Erro
35      } // fim do método main
36
37  } // fim da classe DemoRoboSimplesComErros
```

Os pontos interessantes da listagem 4.8 são:

- A instância `número5`, construída na linha 21 da listagem 4.8, tem como argumentos para seu construtor uma String, dois valores do tipo `char` e outro valor inteiro que foi convertido forçadamente para o tipo `char`. Existe um construtor na classe `RoboSimples` que aceita como argumentos uma String, dois valores do tipo `int` e um do tipo `char` - os argumentos especificados na linha 21 podem ser usados, bastando converter os dois `char` especificados para `int`, o que pode ser feito automaticamente (promoção de tipos de dados). As coordenadas daquela instância serão $(78, 53)$ (valores correspondentes aos códigos dos caracteres passados) e a direção será `'E'` (caracter cujo código é 69).
- A instância `gort` (linha 24) também pode ser criada sem problemas - as coordenadas do tipo `byte` podem ser convertidas para inteiros automaticamente. Apesar da declaração ser similar, a construção da instância `rosie` causará um erro de compilação pois não é possível converter as coordenadas do tipo `float` automaticamente para `int` - o *cast* deve ser feito.
- A instância `data` (linha 28) também não pode ser criada pois não existe nenhum construtor cujo argumento seja um único valor booleano, ou que possam, através da promoção de dados, aceitar o valor booleano (o tipo `boolean` não pode ser promovido nem rebaixado).
- A instância `clonado` (linha 29) também não pode ser criada pois não existe nenhum construtor cujo argumento seja uma própria instância da classe `RoboSimples`. O construtor que aceita uma única String como argumento (declarado na linha 39 da listagem 4.7) não pode ser usado, apesar da classe `RoboSimples` ter o método `toString` que retorna uma String representando uma instância da classe: o método `toString` só é chamado implicitamente quando uma manipulação de Strings é esperada.
- A chamada ao método `move` na linha 32 da listagem 4.8 é válida pois existe na classe `RoboSimples` um método `move` que aceita um inteiro como argumento, e um valor do tipo `byte` pode ser promovido automaticamente sem problemas. A chamada na linha 33 não pode ser feita: o argumento é do tipo `double`, e não poderá ser aceito pelo método que espera um argumento do tipo `int` a não ser que a conversão explícita seja feita.
- A chamada ao método `move` na linha 34 não pode ser feita pois não existe método com assinatura correspondente.

## 4.4 Exercícios do capítulo 4

**Exercício 4.1:** ⋆

Escreva um construtor para a classe `Data` que receba os valores correspondentes ao dia, mês e ano, e inicialize os campos da classe, verificando antes se a data é válida.

**Exercício 4.2:** ⋆

Escreva um construtor para a classe `Lampada` de forma que instâncias desta só possam ser criadas se um estado inicial for passado para o construtor. Este estado pode ser o valor booleano que indica se a lâmpada está acesa (`true`) ou apagada (`false`).

**Exercício 4.3:** ⋆

Considere a classe `Lampada` que também representa o número de watts da lâmpada (veja o exercício 2.21). Escreva dois construtores para a classe: um que recebe como argumentos o número de watts da lâmpada e outro que considera que a lâmpada tem 60 watts por *default*.

**Exercício 4.4:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Ponto2D
2    {
3    private double x,y;
4    ponto2d(double _x,double _y)
5      {
6      x = _x; y = _y;
7      }
8    } // fim da classe
9
```

**Exercício 4.5:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Data
2    {
3    private byte dia,mês;
4    private short ano;
5    private Data(byte d,byte m,short a)
6      {
7      dia = d; mês = m; ano = a;
8      }
9    } // fim da classe
```

**Exercício 4.6:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class IdentidadeDaPessoa
2    {
3    private int númeroDoRG;
4    private int númeroDoCPF;
5    private String nome;
6    void IdentidadeDaPessoa(int nRG,int nCPF,String n)
7      {
8      númeroDoRG = nRG; númeroDoCPF = nCPF; nome = n;
9      }
10   } // fim da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 4.7:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Ponto2D
2    {
3    private double x,y;
4    Ponto2D()
5      {
6      Ponto2D(0.0,0.0);
7      }
8    Ponto2D(double coord1,double coord2)
9      {
10     x = coord1; y = coord2;
11     }
12   } // fim da classe
```

**Exercício 4.8:** ⋆

Liste as assinaturas dos construtores e métodos na classe `RoboSimples` (listagem 4.7).

**Exercício 4.9:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Ponto2D
2    {
3    private double x,y;
4    Ponto2D(double _x,double _y)
5      {
6      x = _x; y = _y;
7      }
8    Ponto2D(double coord1,double coord2)
9      {
10     x = coord1; y = coord2;
11     }
12   } // fim da classe
```

**Exercício 4.10:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class MaisEventosAcademicos
2    {
3    public static void main(String[] argumentos)
4      {
5      EventoAcademico ICFS1999 = new EventoAcademico();
6      EventoAcademico ESBO2000;
7      ESBO2000 = new EventoAcademico("Encontro da Sociedade Brasileira de Origami",
8                                     "Vila Velha (ES)");
9      System.out.println(ICFS1999);
10     System.out.println(ESBO2000);
11     }
12   } // fim da classe
```

**Exercício 4.11:** ⋆

É possível chamar o método `move(1,2)` de uma instância da classe `RoboSimples` ? E os método `move(1.2)` ou `move(true)` ? Porquê ?



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 4.12:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoConstrutor
2    {
3    private int a,b;
4    DemoConstrutor()
5      {
6      System.out.println("No construtor sem argumentos...");
7      DemoConstrutor(0,0);
8      }
9    DemoConstrutor(int xa,int xb)
10     {
11     System.out.println("No construtor com argumentos...");
12     a = xa; b = xb;
13     }
14   } // fim da classe
```

**Exercício 4.13:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Media
2    {
3    public int Media(int a,int b)
4      {
5      return (a+b)/2;
6      }
7    public double Media(int a,int b)
8      {
9      return (a+b)/2;
10     }
11   } // fim da classe
```

**Exercício 4.14:** ⋆

Escreva dois construtores para a classe `Contador`, um que não receba argumentos e considere que o contador começa a contar a partir do zero, e outro que aceita um valor inicial para contagem.

**Exercício 4.15:** ⋆

Considerando a classe `RoboSimples` (listagem 4.7), quais das chamadas ao método `move` abaixo podem ser usadas ? Explique.

- `move();`
- `move(1);`
- `move('A');`
- `move("A");`
- `move(1/3);`
- `move(2,3,5);`
- `move(9,false);`
- `move("17");`
- `move((long)3);`
- `move((char)65);`



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 4.16:** ⋆

Escreva um construtor para a classe `PoligonoRegular` (exercício 2.51), que receba um valor inteiro correspondente ao número de lados do polígono.

**Exercício 4.17:** ⋆⋆

O que aconteceria com a instância `clonado` da classe `RoboSimples` (declarada na linha 30 da listagem 4.8) se a declaração fosse `RoboSimples clonado = new RoboSimples(""+número5);` ?

**Exercício 4.18:** ⋆⋆

Escreva dois construtores para a classe `ContaBancariaSimplificada` (exercício 2.53), um que inicialize todos os campos da classe e outro que considere que o saldo inicial será zero e a conta não será especial.

**Exercício 4.19:** ⋆⋆

Explique, com suas palavras, o que acontecerá se sobrecarregarmos o método `toString`.

**Exercício 4.20:** ⋆⋆

Escreva outro construtor para a classe `Data` que receba uma instância da própria classe `Data` e use os dados desta para inicializar os campos. Veja também o exercício 4.1.

**Exercício 4.21:** ⋆⋆

Suponha que os robôs modelados pela classe `RoboSimples` possam se movimentar para a frente e para trás. Escreva na classe dois métodos `moveParaTrás`, um que mova os robôs uma unidade e outro que aceite um valor como argumento (número de unidades a mover). *Dica:* mover um robô *n* unidades para trás é a mesma coisa que movê-lo *n* unidades para a frente, então podemos chamar o método `move` de dentro do método `moveParaTrás`, trocando o sinal do valor do movimento.

**Exercício 4.22:** ⋆⋆

Escreva três construtores para a classe `NumeroComplexo` (exercício 2.64). Um construtor deverá receber os dois valores (real e imaginário) como argumentos, o outro somente o valor real, considerando o imaginário como sendo zero, e o terceiro construtor não recebe argumentos, considerando as partes real e imaginária do número complexo como sendo iguais a zero.

**Exercício 4.23:** ★★

Considerando as classes abaixo, para cada chamada ao método `doisValores`, identifique que forma do método será chamada.

```
1  class Soma
2    {
3    public int doisValores(int a,int b) // soma dois inteiros
4      {
5      return a+b;
6      }
7    public double doisValores(double a,int b) // soma um double e um inteiro
8      {
9      return a+b;
10     }
11   public double doisValores(double a,double b) // soma dois doubles
12     {
13     return a+b;
14     }
15   }
16
17 class TesteSoma
18   {
19   public static void main(String[] args)
20     {
21     Soma soma = new Soma(); // cria instância da classe Soma
22     // Declara várias variáveis
23     byte b = 20;
24     short s = 99;
25     int i = 1000;
26     long l = 1234L;
27     float f = 3.1416f;
28     double d = 2000;
29     // Chama vários métodos da classe Soma
30     System.out.println(soma.doisValores(b,s));
31     System.out.println(soma.doisValores(i,s));
32     System.out.println(soma.doisValores(i,i));
33     System.out.println(soma.doisValores(l,b));
34     System.out.println(soma.doisValores(f,s));
35     System.out.println(soma.doisValores(d,b));
36     System.out.println(soma.doisValores(b,d));
37     System.out.println(soma.doisValores(i,l));
38     System.out.println(soma.doisValores(l,l));
39     System.out.println(soma.doisValores(d,f));
40     }
41   }
```

**Exercício 4.24:** ★★

Escreva dois construtores para a classe `Ponto2D` (listagem 3.2): um sem argumentos que considere que o ponto está na origem, ou seja, com coordenadas $(0,0)$, e um que receba dois argumento do tipo `double` e que os use para inicializar os campos da classe.

**Exercício 4.25:** ★★★

Escreva uma classe `SiteNaInternet` que represente os dados de um site na Internet, como seu título ou nome, sua URL, a data de primeiro acesso, a data de último acesso e o número de visitas. Use a classe `Contador` (exercício 2.43) para contar o número de visitas, e a classe `DataHora` (exercício 2.47) para representar as datas de primeiro e último acesso. Escreva, para esta classe, o construtor que inicializa todos os campos e o método `toString`.

**Exercício 4.26:** ⋆⋆⋆

Escreva quatro construtores para a classe `Linha` (exercício 2.49): um sem argumentos que considere que a linha começe e termine no ponto $(0,0)$; um que receba um argumento do tipo `Ponto2D` e que considere que a linha começe na origem e termine no ponto passado como argumento; um que receba duas instâncias da classe `Ponto2D` como argumentos e um que receba quatro valores de ponto flutuante, correspondentes às duas coordenadas.

**Exercício 4.27:** ⋆⋆⋆

Escreva quatro construtores para a classe `Retangulo` (exercício 2.50): um sem argumentos que considere que os dois pontos extremos do retângulo tenham coordenadas iguais a $(0,0)$; um que receba um argumento do tipo `Ponto2D` e que considere que um dos pontos extremos do retângulo está na origem do sistema de coordenadas e que o outro seja o ponto passado como argumento; um que receba duas instâncias da classe `Ponto2D` como argumentos e as considere como pontos extremos do retângulo, e um que receba quatro valores de ponto flutuante, correspondentes às duas coordenadas dos pontos extremos.

## 4.5 Exercícios complementares do capítulo 4

**Exercício 4.28:** ⋆

Se a classe `Data` tivesse um método `toString` (pedido no exercício 3.26), o método `toString` da classe `EventoAcademico` (listagem 4.3) poderia ser escrito de maneira muito mais simples. Reescreva o método `toString` da classe `EventoAcademico` usando uma chamada ao método `toString` da classe `Data`.

**Exercício 4.29:** ⋆

Quando chamamos o método `inicializaData` para inicializar campos de uma instância da classe `Data`, somos obrigados a fazer o *cast* para que os argumentos para o método sejam reconhecidos como sendo dos tipos apropriados (vide linhas 35 e 42 da listagem 3.1). Usando sobrecarga, seria possível escrever um outro método `inicializaData` que recebe três argumentos do tipo `int`, que serão convertidos dentro do corpo do método, fazendo com que o programador usuário da classe não precisasse se preocupar com o *cast* em seus programas. Existe um pequeno problema de *design* ou de organização de classes com esta solução, qual é ?

**Exercício 4.30:** ⋆

Modifique o método `inicializaRegistroAcademicoSemConstrutor` da classe `RegistroAcademicoSemConstrutor` para que o valor passado para o campo `percentualDeCobrança` esteja sempre correto (isto é, entre zero e cem). Caso o valor passado seja menor do que zero, deve ser considerado igual a zero, caso o valor seja maior do que cem deve ser considerado como sendo igual a cem.

**Exercício 4.31:** ⋆⋆

O que aconteceria se no construtor da classe `EventoAcademico` as instâncias internas da classe `Data` fossem simplesmente igualadas às instâncias passadas como argumentos (por exemplo, se escrevêssemos `inícioDoEvento = i; fimDoEvento = f;` ao invés do trecho entre as linhas 31 e 34 da listagem 4.3) ? Explique, usando a aplicação na listagem 4.4 como exemplo.

**Exercício 4.32:** ⋆⋆⋆

As coordenadas de posicionamento dos robôs modelados pela classe `RoboSimples` são representadas por valores separados ($x$ e $y$). Reescreva a classe `RoboSimples` de forma que instâncias da classe `Ponto2D` sejam usadas para representar as coordenadas dos robôs. Para isto, reescreva também a própria classe `Ponto2D` para que seja possível modificar as coordenadas encapsuladas.

# Capítulo 5

# Campos e Métodos Estáticos

## 5.1 Introdução

Vimos nos capítulos anteriores que cada instância de uma classe terá uma cópia de todos os campos declarados na classe - esta é uma das características mais úteis de linguagens de programação orientadas a objeto: a possibilidade da criação de novos "tipos de dados" que são compostos de outros dados convenientemente agrupados em uma única entidade.

As instâncias de uma classe são, então, independentes entre si: a modificação do campo `dia` de uma instância da classe `Data` não afeta o valor do mesmo campo em outra instância. Apesar de esperado, nem sempre este comportamento é desejável: não foi visto, até agora, um mecanismo que permita o compartilhamento de informações entre todas as instâncias de uma classe, o que poderia ser usado, por exemplo, para algum tipo de controle interno.

Neste capítulo veremos como podemos declarar campos que serão compartilhados entre todas as instâncias de uma mesma classe em uma aplicação, assim como a criação de métodos que não precisam de instâncias de classes para ser executados.

## 5.2 Campos estáticos em classes

*campos estáticos*

Campos estáticos em uma classe são compartilhados por todas as instâncias desta classe - em outras palavras, somente um valor será armazenado em um campo estático, e caso este valor seja modificado por uma das instâncias desta classe, a modificação será refletida em todas as outras instâncias desta classe.

*modificador* `static`

Campos estáticos são declarados com o modificador `static`, que deve ser declarado antes do tipo de dado do campo e pode ser combinado com modificadores de acesso como `public` e `private`.

*campos de classes*

*campos de instâncias*

Campos estáticos são também conhecidos como *campos de classes*, já que estes campos poderão ser acessados diretamente usando o nome da classe, sem que seja necessária a criação de uma instância da classe e uma referência para esta instância. Em contraste, campos que podem ter diferentes valores para cada instância da mesma classe (como os vistos nos exemplos anteriores) são conhecidos como *campos de instâncias*.

*constantes*

As duas maiores utilidades de campos estáticos em classes são manter uma informação ou estado para todas as instâncias de uma classe que possa ser modificada ou acessada por qualquer das instâncias e armazenar valores que não serão modificados nem serão variáveis por instâncias de

classes, isto é, *valores constantes*.

Como exemplo de informação que deve ser compartilhada por todas as instâncias de uma classe, consideremos o problema da simulação de um caixa de banco. Suponhamos que neste banco não exista um sistema de senhas ou fila única, então cada caixa é independente dos outros em relação ao atendimento do cliente, isto é, cada caixa terá que ter sua própria fila, e os clientes serão atendidos à medida em que o cliente anterior sair da fila. Para ilustrar, a figura 5.1 mostra, à esquerda, um banco com cinco caixas, cada um com sua fila, e à direita, um banco com cinco caixas e fila única.

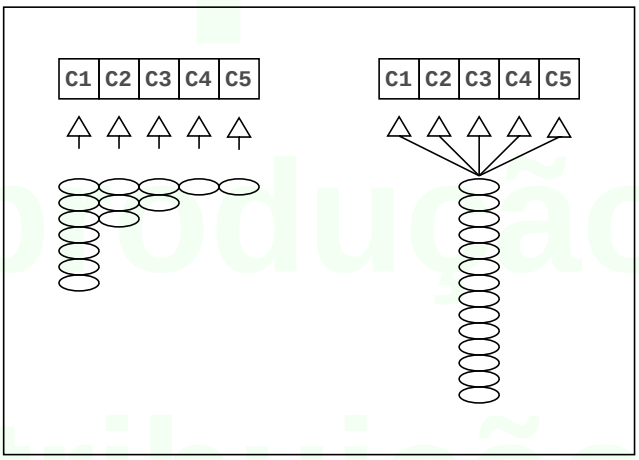



Figura 5.1: Bancos com e sem fila única

Para demonstração do uso de campos estáticos que representem um único valor para todas as instâncias de uma classe, vamos criar classes que simulem o atendimento de caixas em bancos com e sem fila única. Uma classe que simula o atendimento de um caixa de banco com fila única é mostrada na listagem 5.1.

Listagem 5.1: A classe `SimuladorDeCaixaDeBanco0`, que simula o mecanismo de atendimento de um caixa de banco.

```
1  /**
2   * A classe SimuladorDeCaixaDeBanco0, que simula um caixa de banco, onde o atendimento
3   * é feito através de uma senha que indica a ordem de atendimento. O nome da classe tem
4   * o número 0 porque esta classe será melhorada posteriormente.
5   */
6  class SimuladorDeCaixaDeBanco0 // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10   */
11   private int númeroDoCliente; // o número do cliente a ser atendido
12   private int númeroDoCaixa; // o número do caixa (para sua identificação)
13
14  /**
15   * O construtor da classe inicializa o número do caixa (para identificação),
16   * inicializa o contador de clientes e imprime uma mensagem.
17   * @param n o número do caixa dentro do banco
18   */
19   SimuladorDeCaixaDeBanco0(int n)
20     {
21     númeroDoCaixa = n;
22     númeroDoCliente = 0;
23     System.out.println("Caixa "+númeroDoCaixa+" iniciou operação.");
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
24      } // fim do construtor
25
26    /**
27     * O método próximoAtendimento simula o atendimento de um cliente.
28     */
29    public void iniciaAtendimento()
30      {
31      númeroDoCliente = númeroDoCliente + 1; // o próximo cliente será chamado
32      System.out.print("Cliente com a senha número "+númeroDoCliente+", favor ");
33      System.out.println("dirigir-se ao caixa número "+númeroDoCaixa+".");
34      }
35
36    } // fim da classe SimuladorDeCaixaDeBanco0
```

A demonstração do funcionamento da classe `SimuladorDeCaixaDeBanco0` através da criação e uso de várias instâncias da classe é feita pelo programa na listagem 5.2.

Listagem 5.2: A classe `DemoSimuladorDeCaixaDeBanco0`, que demonstra usos de instâncias da classe `SimuladorDeCaixaDeBanco0`.

```
1  /**
2   * A classe DemoSimuladorDeCaixaDeBanco0, que demonstra usos de instâncias da classe
3   * SimuladorDeCaixaDeBanco0, em especial, o problema que ocorre por causa da
4   * independência das instâncias da classe SimuladorDeCaixaDeBanco0.
5   */
6  class DemoSimuladorDeCaixaDeBanco0 // declaração da classe
7    {
8    /**
9     * O método main permite a execução desta classe. Este método contém declarações
10    * de algumas instâncias da classe SimuladorDeCaixaDeBanco0, e demonstra sua criação
11    * e uso.
12    * @param argumentos os argumentos que podem ser passados para o método via linha
13    *        de comando, mas que neste caso serão ignorados.
14    */
15   public static void main(String[] argumentos)
16     {
17     // Declaramos cinco referências a instâncias da classe SimuladorDeCaixaDeBanco0,
18     // e as usamos para simular os caixas de um banco.
19     SimuladorDeCaixaDeBanco0 c1 = new SimuladorDeCaixaDeBanco0(1);
20     SimuladorDeCaixaDeBanco0 c2 = new SimuladorDeCaixaDeBanco0(2);
21     SimuladorDeCaixaDeBanco0 c3 = new SimuladorDeCaixaDeBanco0(3);
22     SimuladorDeCaixaDeBanco0 c4 = new SimuladorDeCaixaDeBanco0(4);
23     SimuladorDeCaixaDeBanco0 c5 = new SimuladorDeCaixaDeBanco0(5);
24     // Fazemos várias simulações de atendimento com balanceamento desigual.
25     c1.iniciaAtendimento();
26     c2.iniciaAtendimento();
27     c3.iniciaAtendimento();
28     c4.iniciaAtendimento();
29     c5.iniciaAtendimento();
30     c1.iniciaAtendimento();
31     c2.iniciaAtendimento();
32     c3.iniciaAtendimento();
33     c1.iniciaAtendimento();
34     c2.iniciaAtendimento();
35     c1.iniciaAtendimento();
36     c1.iniciaAtendimento();
37     c1.iniciaAtendimento();
38     c1.iniciaAtendimento();
39     } // fim do método main
40
41   } // fim da classe DemoSimuladorDeCaixaDeBanco0
```

O resultado da execução do programa na listagem 5.2 é mostrado a seguir:

```
1  Caixa 1 iniciou operação.
2  Caixa 2 iniciou operação.
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
Caixa 3 iniciou operação.
Caixa 4 iniciou operação.
Caixa 5 iniciou operação.
Cliente com a senha número 1, favor dirigir-se ao caixa número 1.
Cliente com a senha número 1, favor dirigir-se ao caixa número 2.
Cliente com a senha número 1, favor dirigir-se ao caixa número 3.
Cliente com a senha número 1, favor dirigir-se ao caixa número 4.
Cliente com a senha número 1, favor dirigir-se ao caixa número 5.
Cliente com a senha número 2, favor dirigir-se ao caixa número 1.
Cliente com a senha número 2, favor dirigir-se ao caixa número 2.
Cliente com a senha número 2, favor dirigir-se ao caixa número 3.
Cliente com a senha número 3, favor dirigir-se ao caixa número 1.
Cliente com a senha número 3, favor dirigir-se ao caixa número 2.
Cliente com a senha número 4, favor dirigir-se ao caixa número 1.
Cliente com a senha número 5, favor dirigir-se ao caixa número 1.
Cliente com a senha número 6, favor dirigir-se ao caixa número 1.
Cliente com a senha número 7, favor dirigir-se ao caixa número 1.
```

Como podemos ver através das listagens 5.1 e 5.2 e do resultado da aplicação, cada instância da classe `SimuladorDeCaixaDeBanco0` tem seus próprios campos, o que faz com que a modificação do campo `númeroDoCliente` de uma instância da classe (causada pelo atendimento a um cliente) não modifique os valores do mesmo campo em outras instâncias - desta forma, cada caixa do banco é independente, e os clientes teriam senhas que só valeriam para aquele caixa.

Para simular caixas em um banco que compartilhassem informações - no caso, qual é o número da senha do próximo cliente a ser atendido, que deve ser um único valor que pode ser visto e modificado por todos os caixas - poderíamos tentar passar a informação para o próximo método a ser chamado, de forma que o método `iniciaAtendimento` recebesse como argumento o número da senha do cliente e retornasse este valor mais um, ou seja, o próximo número de senha.

Esta abordagem simplista tem dois problemas em potencial: o controle de quem será o próximo a ser atendido no simulador fica a cargo do programador usuário - o ideal seria encapsular o comportamento do caixa do banco na classe, e não repassar este comportamento para programas que usem a classe. Adicionalmente, não existe controle real da seqüência de atendimento - se o número da próxima senha fosse passado como argumento, qualquer número poderia ser passado, comprometendo o funcionamento do simulador.

Claramente um outro mecanismo de compartilhamento de informações entre instâncias da mesma classe na mesma aplicação seria melhor. Java permite que campos em uma classe sejam modificados para ser considerados estáticos - independentemente de quantas instâncias sejam criadas, somente um valor será armazenado para cada campo declarado como estático. No nosso exemplo, basta declarar o campo `númeroDoCliente` como sendo estático que ele sempre armazenará o último número de senha para todas as instâncias da classe. A classe reescrita (e com o nome modificado) é mostrada na listagem 5.3.

Listagem 5.3: A classe `SimuladorDeCaixaDeBanco`, que simula o mecanismo de atendimento de um caixa de banco (com melhorias).

```
/**
 * A classe SimuladorDeCaixaDeBanco, que simula um caixa de banco, onde o atendimento
 * é feito através de uma senha que indica a ordem de atendimento. Um dos campos desta
 * classe será declarado como estático, significando que somente um valor será
 * armazenado neste campo, independente de quantas instâncias da classe sejam criadas
```

```
 * em um programa.
 */
class SimuladorDeCaixaDeBanco // declaração da classe
  {
 /**
  * Declaração dos campos da classe
  */
  static private int númeroDoCliente; // o número do cliente a ser atendido. Este campo
                                      // é declarado estático, somente um único valor
                                      // poderá ser armazenado nele independentemente
                                      // de quantas instâncias desta classe sejam
                                      // criadas.
  private int númeroDoCaixa; // o número do caixa (para sua identificação)

 /**
  * O construtor da classe inicializa o número do caixa (para identificação),
  * inicializa o contador de clientes e imprime uma mensagem.
  * @param n o número do caixa dentro do banco
  */
  SimuladorDeCaixaDeBanco(int n)
    {
    númeroDoCaixa = n;
    númeroDoCliente = 0;
    System.out.println("Caixa "+númeroDoCaixa+" iniciou operação.");
    } // fim do construtor

 /**
  * O método próximoAtendimento simula o atendimento de um cliente.
  */
  public void iniciaAtendimento()
    {
    númeroDoCliente = númeroDoCliente + 1; // o próximo cliente será chamado
    System.out.print("Cliente com a senha número "+númeroDoCliente+", favor ");
    System.out.println("dirigir-se ao caixa número "+númeroDoCaixa+".");
    }

  } // fim da classe SimuladorDeCaixaDeBanco
```

Uma aplicação que usa instâncias da classe `SimuladorDeCaixaDeBanco` pode ser reescrita a partir da listagem 5.2, trocando-se as instâncias da classe `SimuladorDeCaixaDeBanco0` por instâncias da classe `SimuladorDeCaixaDeBanco`. O resultado de uma aplicação que usa instâncias da classe `SimuladorDeCaixaDeBanco` é mostrado a seguir:

```
Caixa 1 iniciou operação.
Caixa 2 iniciou operação.
Caixa 3 iniciou operação.
Caixa 4 iniciou operação.
Caixa 5 iniciou operação.
Cliente com a senha número 1, favor dirigir-se ao caixa número 1.
Cliente com a senha número 2, favor dirigir-se ao caixa número 2.
Cliente com a senha número 3, favor dirigir-se ao caixa número 3.
Cliente com a senha número 4, favor dirigir-se ao caixa número 4.
Cliente com a senha número 5, favor dirigir-se ao caixa número 5.
Cliente com a senha número 6, favor dirigir-se ao caixa número 1.
Cliente com a senha número 7, favor dirigir-se ao caixa número 2.
Cliente com a senha número 8, favor dirigir-se ao caixa número 3.
Cliente com a senha número 9, favor dirigir-se ao caixa número 1.
Cliente com a senha número 10, favor dirigir-se ao caixa número 2.
Cliente com a senha número 11, favor dirigir-se ao caixa número 1.
Cliente com a senha número 12, favor dirigir-se ao caixa número 1.
Cliente com a senha número 13, favor dirigir-se ao caixa número 1.
```

```
19 Cliente com a senha número 14, favor dirigir-se ao caixa número 1.
```

Com este resultado, podemos ver que as instâncias da classe `SimuladorDeCaixaDeBanco.java` compartilham o mesmo valor através do campo estático `númeroDoCliente`.

Um outro uso de campos estáticos em classes é o de criação de constantes, que serão compartilhadas por todas as instâncias da classe e, em geral, imutáveis. Constantes em classes geralmente são acessadas através da classe e não de instâncias desta classe. Como exemplo, consideremos a classe `ConstantesMatematicas`, mostrada na listagem 5.4.

Listagem 5.4: A classe `ConstantesMatematicas`, que contém algumas constantes matemáticas úteis.

```
1  /**
2   * A classe ConstantesMatematicas, que contém vários valores que são constantes.
3   * Esta classe não tem métodos, e todos os campos são declarados como static, assim
4   * não é necessário criar instâncias desta classe para acessar os valores. Para que
5   * os valores possam ser acessados de fora da classe sem a necessidade de métodos os
6   * campos são declarados como public. Para garantir que os valores não poderão ser
7   * modificados, os campos também são declarados como final.
8   */
9  class ConstantesMatematicas // declaração da classe
10   {
11  /**
12   * Declaração dos campos da classe
13   */
14   // A raiz quadrada de 2
15   final static public double raizDe2 = 1.4142135623730950488;
16   // A raiz quadrada de 3
17   final static public double raizDe3 = 1.7320508075688772935;
18   // A raiz quadrada de 5
19   final static public double raizDe5 = 2.2360679774997896964;
20   // A raiz quadrada de 6: podemos usar as constantes já definidas
21   final static public double raizDe6 = raizDe2*raizDe3;
22   } // fim da classe ConstantesMatematicas
23
24 // Valores obtidos no livro Manual de Fórmulas e Tabelas Matemáticas, Murray R.
25 // Spiegel, Coleção Schaum, editora McGraw-Hill
```

*modificador* `final`

Um ponto importante na listagem 5.4 é que as constantes, que são declaradas com o modificador `static` para que sejam os mesmo valores independentemente de quantas instâncias da classe sejam criadas, também são declaradas com o modificador `final`. O modificador `final` faz com que os valores dos campos, depois de declarados, não possam mais ser modificados, o que é desejável para campos que representam constantes. A aplicação na listagem 5.5 demonstra usos das constantes na classe `ConstantesMatematicas`.

Listagem 5.5: A classe `DemoConstantesMatematicas`, que demonstra o uso dos campos estáticos na classe `ConstantesMatematicas`.

```
1  /**
2   * A classe DemoConstantesMatematicas, que demonstra usos da classe
3   * ConstantesMatematicas, em especial, o acesso aos campos estáticos da classe.
4   */
5  class DemoConstantesMatematicas // declaração da classe
6    {
7    /**
8     * O método main permite a execução desta classe. Este método demonstra o acesso a
9     * campos estáticos da classe ConstantesMatematicas, através de instâncias e através
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
10    * do acesso direto.
11    * @param argumentos os argumentos que podem ser passados para o método via linha
12    *        de comando, mas que neste caso serão ignorados.
13    */
14   public static void main(String[] argumentos)
15     {
16     // Criamos duas instâncias da classe ConstantesMatematicas. Como os campos desta
17     // classe são estáticos, os valores são idênticos independentemente das instâncias.
18     ConstantesMatematicas const1 = new ConstantesMatematicas();
19     ConstantesMatematicas const2 = new ConstantesMatematicas();
20     // Verificamos a igualdade...
21     System.out.println(const1.raizDe2 == const2.raizDe2); // imprime true
22     System.out.println(const1.raizDe3 == const2.raizDe3); // imprime true
23     System.out.println(const1.raizDe5 == const2.raizDe5); // imprime true
24     System.out.println(const1.raizDe6 == const2.raizDe6); // imprime true
25     // É muito mais prático acessar os campos diretamente a partir da classe:
26     double raizDe10 = ConstantesMatematicas.raizDe2 * ConstantesMatematicas.raizDe5;
27     System.out.println("A raiz quadrada de 10 é "+raizDe10);
28     } // fim do método main
29
30   } // fim da classe DemoConstantesMatematicas
```

Podemos ver na listagem 5.5 que os valores constantes são iguais para todas as instâncias da classe `ConstantesMatematicas`, e que a criação de instâncias de uma classe onde todos os campos são declarados como `static` não é necessária: podemos acessar os campos diretamente (já que eles são declarados como `public`) através do nome da classe.

## 5.3 Métodos estáticos em classes

*métodos estáticos*

*métodos de classes*

Métodos estáticos em uma classe também são declarados com o modificador `static`, que deve preceder o tipo de retorno do método e que pode ser combinado com modificadores de acesso ao método. A diferença principal entre métodos estáticos (também conhecidos como *métodos de classes*) e métodos não-estáticos é que os métodos estáticos podem ser chamados sem a necessidade de criação de instâncias das classes às quais pertencem.

No capítulo 3 já vimos um método estático - o método `main` - que permite que uma classe seja executada como uma aplicação ou programa. Se uma classe, por exemplo, `Teste` tem o método estático `main` declarado adequadamente, não precisamos criar instâncias de `Teste` para executar o método `main`. Métodos estáticos podem ser usados em classes que tenham o método `main` para servirem como subrotinas deste, o que será descrito na seção 5.4.

Métodos estáticos são também adequados para implementar rotinas que sejam independentes de dados armazenados em classes, ou seja, métodos que só necessitem dos dados passados como argumentos para efetuar a tarefa requerida, e que sejam executados com o mesmo resultado independentemente de qual instância da classe a que pertencem seja usada para sua chamada.

*bibliotecas de métodos*

A aplicação mais frequente de métodos estáticos é a criação de *bibliotecas de métodos*, classes que contém somente métodos estáticos, geralmente agrupados por função. Um exemplo de uma biblioteca de métodos (que tem métodos para calcular várias conversões de unidades de comprimento) é mostrada na listagem 5.6.

Listagem 5.6: A classe `ConversaoDeUnidadesDeComprimento`, que contém métodos estáticos para a conversão de unidades.

```
/**
 * A classe ConversaoDeUnidadesDeComprimento, que contém vários métodos estáticos que
 * convertem unidades de comprimento. Esta classe não tem campos, e todos os seus
 * métodos são declarados como static, assim não é necessário criar instâncias desta
 * classe para usar os métodos. Para que os métodos possam ser acessados de qualquer
 * outra classe eles são declarados como public.
 */
class ConversaoDeUnidadesDeComprimento // declaração da classe
  {
 /**
  * O método polegadasParaCentímetros converte o valor passado em polegadas para
  * centímetros.
  * @param polegadas o número de polegadas
  * @return o número de centímetros correspondente ao número de polegadas
  */
  public static double polegadasParaCentímetros(double polegadas)
    {
    double centímetros = polegadas*2.54;
    return centímetros;
    }

 /**
  * O método pésParaCentímetros converte o valor passado em pés para centímetros.
  * @param pés o número de pés
  * @return o número de centímetros correspondente ao número de pés
  */
  public static double pésParaCentímetros(double pés)
    {
    double centímetros = pés*30.48;
    return centímetros;
    }

 /**
  * O método milhasParaQuilômetros converte o valor passado em milhas para quilômetros.
  * @param milhas o número de milhas
  * @return o número de quilômetros correspondente ao número de milhas
  */
  public static double milhasParaQuilômetros(double milhas)
    {
    double quilômetros = milhas*1.609;
    return quilômetros;
    }

  } // fim da classe ConversaoDeUnidadesDeComprimento

// Fórmulas obtidas no livro Manual de Fórmulas e Tabelas Matemáticas, Murray R.
// Spiegel, Coleção Schaum, editora McGraw-Hill
```

Os métodos na classe `ConversaoDeUnidadesDeComprimento` são demonstrados na aplicação na listagem 5.7.

Listagem 5.7: A classe `DemoConversaoDeUnidadesDeComprimento`, que demonstra o uso dos métodos estáticos na classe `ConversaoDeUnidadesDeComprimento`.

```
/**
 * A classe DemoConversaoDeUnidadesDeComprimento, que demonstra usos dos métodos
 * estáticos da classe ConversaoDeUnidadesDeComprimento.
 */
class DemoConversaoDeUnidadesDeComprimento // declaração da classe
  {
  /**
   * O método main permite a execução desta classe. Este método demonstra o uso de
   * métodos estáticos da classe ConversaoDeUnidadesDeComprimento, através da criação
   * de instâncias e através do acesso direto.
   * @param argumentos os argumentos que podem ser passados para o método via linha
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
12    *        de comando, mas que neste caso serão ignorados.
13    */
14   public static void main(String[] argumentos)
15     {
16     // Criamos uma instância da classe ConversaoDeUnidadesDeComprimento. Como a classe
17     // não contém campos e os métodos são estáticos, não existe real diferença entre
18     // chamar os métodos de uma ou outra instância da classe.
19     ConversaoDeUnidadesDeComprimento conv = new ConversaoDeUnidadesDeComprimento();
20     // Vamos converter alguns valores:
21     System.out.println("vinte pés são "+conv.pésParaCentímetros(20)+
22                        " centímetros");
23     System.out.println("cinco polegadas são "+conv.polegadasParaCentímetros(5)+
24                        " centímetros");
25     // É muito mais prático acessar os métodos diretamente a partir da classe:
26     System.out.println("vinte pés são "+
27                        ConversaoDeUnidadesDeComprimento.pésParaCentímetros(20)+
28                        " centímetros");
29     System.out.println("cinco polegadas são "+
30                        ConversaoDeUnidadesDeComprimento.polegadasParaCentímetros(5)+
31                        " centímetros");
32     } // fim do método main
33
34   } // fim da classe DemoConversaoDeUnidadesDeComprimento
```

Podemos ver na aplicação mostrada na listagem 5.7 que não é necessário criar uma instância da classe `ConversaoDeUnidadesDeComprimento` para acessar seus métodos, embora isto seja possível. Curiosamente, a instância `conv`, declarada na linha 20 da listagem acima, poderia ser inicializada com o valor `null` e o programa ainda seria executado normalmente - o que importa neste caso é que a instância é da classe `ConversaoDeUnidadesDeComprimento`, independente de ser inicializada ou não.

*variáveis em métodos estáticos*

Variáveis que forem declaradas dentro de métodos estáticos serão automaticamente estáticas, mas como seu escopo é somente dentro do método elas não poderão ser acessadas a partir de outros métodos.

## 5.4 Campos e métodos estáticos em aplicações

Conforme explicado no capítulo 3, o método `main`, quando declarado apropriadamente, será o ponto de entrada que permite a execução de uma classe. O método `main` deve ser declarado como estático para que não seja necessário instanciar a classe a que pertence.

*método local*

*subrotina*

Eventualmente será interessante que o método `main` possa chamar um outro método que não faça parte de outra classe, mas esteja contido na mesma (um *método local*, mais conhecido como *subrotina*), geralmente para simplificar o código (embora seja aconselhável, sempre que seja possível, agrupar vários métodos com características em comum em uma classe, que funcionará como uma biblioteca de rotinas).

Se um método for chamado diretamente a partir do método `main`, este método deverá ser obrigatoriamente declarado como estático. Se o método `main` for acessar campos declarados na sua classe mas fora do método `main`, estes campos também deverão ser declarados como estáticos.

Partes do código que podem ser executadas repetidamente são candidatas à criação de métodos locais, como mostrado na aplicação na listagem 5.8.

Listagem 5.8: A classe `CalculoDePrecoDeTerreno`, que calcula o preço de um terreno baseado em sua área e localização, usando um método estático.

```
/**
 * A classe CalculoDePrecoDeTerreno, que calcula o preço de um terreno baseado
 * em sua área e localização. O cálculo é feito por um método estático da classe,
 * permitindo o seu reuso.
 */
class CalculoDePrecoDeTerreno // declaração da classe
  {
  /**
   * O método main permite a execução desta classe. Este método demonstra o uso de um
   * método estático para calcular o preço de um terreno baseado na sua área e
   * localização. O cálculo é feito usando um método estático nesta classe (subrotina).
   * @param argumentos os argumentos que podem ser passados para o método via linha
   *        de comando, mas que neste caso serão ignorados.
   */
  public static void main(String[] argumentos)
    {
    double preço;
    // Cálculo do preço do terreno no lote N1
    System.out.print("O preço do terreno N1 é ");
    preço = preçoDoTerreno(450,1); // calculamos o preço e o armazenamos na variável
    System.out.println(preço); // imprimimos a variável
    // Cálculo do preço do terreno no lote Q2
    System.out.print("O preço do terreno Q2 é ");
    preço = preçoDoTerreno(475,4);
    System.out.println(preço);
    // Cálculo do preço do terreno no lote F3
    System.out.print("O preço do terreno F3 é ");
    // Chamamos o método e imprimimos o seu resultado
    System.out.println(preçoDoTerreno(525,2));
    } // fim do método main

  /**
   * O método preçoDoTerreno calcula o preço de um terreno dependendo da sua área
   * em metros quadrados e sua localização, que é um código entre 1 e 5.
   * @param área a área do terreno em metros quadrados
   * @param localização o código da localização do terreno (1 a 5)
   * @return o preço do terreno
   */
  private static double preçoDoTerreno(double área,int localização)
    {
    double preço = 0; // deve ser inicializada com algum valor !
    if (localização == 1) preço = área*22.00;
    if (localização == 2) preço = área*27.00;
    if (localização == 3) preço = área*29.50;
    if (localização == 4) preço = área*31.50;
    if (localização == 5) preço = área*34.50;
    return preço;
    } // fim do método preçoDoTerreno

  } // fim da classe CalculoDePrecoDeTerreno
```

Curiosamente, o próprio método `main` pode ser chamado a partir de outros métodos, inclusive de métodos `main` de outras classes. Desta forma uma aplicação inteira (que seria executada pelo método `main` de uma classe) pode ser considerada como uma subrotina de outra aplicação.

*método* `main` *como subrotina*

A listagem 5.9 mostra como métodos estáticos (`main` e outros) de outras classes podem ser chamados a partir do método `main`.

Listagem 5.9: A classe `DemoChamadaAoMain`, que mostra como métodos estáticos de outras classes podem ser chamados a partir do método `main` de uma classe.

```
/**
 * A classe DemoChamadaAoMain, que mostra como métodos estáticos (main e outros) de
 * outras classes podem ser chamados a partir do método main de uma classe.
 */
class DemoChamadaAoMain // declaração da classe
  {
  /**
   * O método main permite a execução desta classe. Este método demonstra como outros
   * métodos main, de outras classes, podem ser chamados a partir deste.
   * @param argumentos os argumentos que podem ser passados para o método via linha
   *        de comando, mas que neste caso serão ignorados.
   */
  public static void main(String[] argumentos)
    {
    // Executamos o método main da classe DemoConstantesMatematicas como se fosse uma
    // subrotina. Como todos os métodos main esperam um array de Strings como
    // argumento, podemos simplesmente repassar os argumentos deste método main para
    // o da classe DemoConstantesMatematicas.
    DemoConstantesMatematicas.main(argumentos);
    // Executamos o método main da classe DemoConversaoDeUnidadesDeComprimento como
    // se fosse uma subrotina.
    DemoConversaoDeUnidadesDeComprimento.main(argumentos);
    // Para demonstrar a chamada de métodos estáticos em outras classes, vamos
    // calcular o preço de um terreno:
    System.out.print("O preço do terreno J1 é ");
    // Chamamos o método e imprimimos o seu resultado
    System.out.println(CalculoDePrecoDeTerreno.preçoDoTerreno(600,5));
    } // fim do método main

  } // fim da classe DemoChamadaAoMain
```

## 5.5 Fábricas de instâncias

*fábricas de instâncias*

Métodos estáticos que retornem novas instâncias de classes são conhecidos como *fábricas de instâncias*. Vimos algo semelhante com o método `origem` (que não era estático) da classe `Ponto2D` (listagem 3.2). Fábricas de instâncias são úteis para a criação simples e rápida de instâncias que sejam bem características de uma classe.

Nem sempre a criação destas fábricas é justificável. Por exemplo, considerando ainda a classe `Ponto2D`, podemos assumir que uma instância desta classe representando a origem do sistema de coordenadas é comum, e sua criação relativamente frequente - desta forma a criação de uma fábrica de instâncias que retornasse uma instância com a origem do sistema de coordenadas seria justificável. Por outro lado, considerando a classe `Aluno`, nenhum conjunto de dados que represente um aluno em particular é tão comum que justificasse a criação de uma fábrica de instâncias da classe `Aluno`.

A listagem 5.10 mostra uma classe que implementa uma fábrica de instâncias dela própria. É interessante notar que podemos ter fábricas de instâncias de classes declaradas em outras classes.

Listagem 5.10: A classe `DataComFabrica`, que contém uma fábrica de instâncias da própria classe.

```
/**
 * A classe DataComFabrica, que contém uma fábrica de instâncias da própria classe.
 * Esta classe contém campos que representam uma data e métodos simples para a
 * manipulação destes campos. Esta classe contém também o construtor (que não verifica
```

```
 * a validade dos dados) e o método toString.
 */
class DataComFabrica // declaração da classe
  {
 /**
  * Declaração dos campos da classe
  */
  private byte dia,mês; // dia e mês são representados por bytes
  private short ano; // ano é representado por um short

 /**
  * O construtor da classe DataComFabrica recebe argumentos para inicializar os campos
  * da classe. Este construtor não verifica a validade da data para manter a classe
  * simples.
  * @param d o argumento correspondente ao método dia
  * @param m o argumento correspondente ao método mês
  * @param a o argumento correspondente ao método ano
  */
  DataComFabrica(byte d,byte m,short a)
    {
    dia = d; mês = m; ano = a;
    } // fim do construtor

 /**
  * O método toString retorna uma String contendo os valores dos campos formatados
  * @return uma String com a data formatada
  */
  public String toString()
    {
    return dia+"/"+mês+"/"+ano;
    }  // fim do método toString

 /**
  * O método Natal (uma fábrica de instâncias da classe Data) retorna o dia de Natal
  * para o ano passado como argumento.
  * @param ano o ano para o qual retornaremos a data de Natal
  * @return uma instância da classe Data correspondente ao Natal daquele ano
  */
  public static DataComFabrica Natal(short ano)
    {
    return new DataComFabrica((byte)25,(byte)12,ano);
    }  // fim do método Natal

  } // fim da classe DataComFabrica
```

A classe `DataComFabrica`, mostrada na listagem 5.10, contém um método estático que retorna uma nova instância da própria classe contendo a data do Natal de um determinado ano, que deve ser passado como argumento para este método. A classe `DemoDataComFabrica`, mostrada na listagem 5.11, demonstra usos deste método.

Listagem 5.11: A classe `DemoDataComFabrica`, que demonstra usos da classe `DataComFabrica`.

```
/**
 * A classe DemoDataComFabrica, que demonstra usos da classe DataComFabrica.
 */
class DemoDataComFabrica // declaração da classe
  {
  /**
   * O método main permite a execução desta classe. Este método cria algumas
   * instâncias da classe DataComFabrica usando a fábrica de instâncias desta
   * classe.
   * @param argumentos os argumentos que podem ser passados para o método via linha
   *        de comando, mas que neste caso serão ignorados.
   */
  public static void main(String[] argumentos)
```

```
14      {
15      // Declaramos algumas referências à instâncias da classe DataComFabrica que serão
16      // inicializadas através do método Natal (a fábrica de instâncias).
17      DataComFabrica NatalDe1966 = DataComFabrica.Natal((short)1966);
18      DataComFabrica NatalDe1970 = DataComFabrica.Natal((short)1970);
19      DataComFabrica NatalDe2000 = DataComFabrica.Natal((short)2000);
20      // Imprimimos as datas através da chamada implícita ao método toString
21      System.out.println(NatalDe1966);
22      System.out.println(NatalDe1970);
23      System.out.println(NatalDe2000);
24      } // fim do método main
25
26    } // fim da classe DemoDataComFabrica
```

Podemos ver que a criação de instâncias da classe `DataComFabrica` pode ser feita de maneira simples com a fábrica de instâncias.

## 5.6 Exercícios do capítulo 5

**Exercício 5.1:** ⋆

Explique, com suas palavras, porque os campos na classe `ConstantesMatematicas` (listagem 5.4) não devem ser declarados com o modificador `private`.

**Exercício 5.2:** ⋆

Escreva, para a classe `DataComFabrica` (listagem 5.10), um método `primeiroDeAbril` que se comporte como uma fábrica de instâncias.

**Exercício 5.3:** ⋆

Escreva, para a classe `DataComFabrica` (listagem 5.10), um método `seteDeSetembro` que se comporte como uma fábrica de instâncias.

**Exercício 5.4:** ⋆

Escreva, para a classe `DataComFabrica` (listagem 5.10), um método `primeiroDoMês` que se comporte como uma fábrica de instâncias.

**Exercício 5.5:** ⋆⋆

O método `main` pode ser chamado a partir de outro método da mesma classe. Se isto for feito, que problemas podem ocorrer na aplicação ?

**Exercício 5.6:** ⋆⋆

Adicione à classe `ConversaoDeUnidadesDeComprimento` (listagem 5.6) mais métodos estáticos para conversão de unidades. Considere a lista de medidas abaixo como referência, e crie métodos para fazer as respectivas conversões.

- 1 centímetro = 0.3937 polegadas
- 1 metro = 39.37 polegadas
- 1 quilômetro = 0.6214 milhas



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 5.7:** ⋆⋆

Escreva a classe `ConversaoDeUnidadesDeArea` com métodos estáticos para conversão das unidades de área segundo a lista abaixo.

- 1 metro quadrado = 10.76 pés quadrados
- 1 pé quadrado = 929 centímetros quadrados
- 1 milha quadrada = 640 acres
- 1 acre = 43.560 pés quadrados

**Exercício 5.8:** ⋆⋆

Escreva a classe `ConversaoDeUnidadesDeVolume` com métodos estáticos para conversão das unidades de volume segundo a lista abaixo.

- 1 litro = 1000 centímetros cúbicos
- 1 metro cúbico = 1000 litros
- 1 metro cúbico = 35.32 pés cúbicos
- 1 galão americano = 231 polegadas cúbicas
- 1 galão americano = 3.785 litros

**Exercício 5.9:** ⋆⋆

Escreva a classe `ConversaoDeUnidadesDeVelocidade` com métodos estáticos para conversão das unidades de velocidade segundo a lista abaixo.

- 1 quilômetro por hora = 0.2778 metros por segundo
- 1 quilômetro por hora = 0.6214 milhas por hora
- 1 quilômetro por hora = 0.9113 pés por segundo
- 1 milha por hora = 1.467 pés por segundo
- 1 milha por hora = 1.609 quilômetros por hora
- 1 milha por hora = 0.4470 metros por segundo

**Exercício 5.10:** ⋆⋆

Escreva a classe `ConversaoDeUnidadesDeTempo` com métodos estáticos para conversão aproximada das unidades de velocidade segundo a lista abaixo.

- 1 minuto = 60 segundos
- 1 hora = 60 minutos
- 1 dia = 24 horas
- 1 semana = 7 dias
- 1 mês = 30 dias
- 1 ano = 365.25 dias



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 5.11:** ⋆⋆

Escreva uma classe `ConversaoDeTemperatura` que contenha métodos estáticos para calcular a conversão entre diferentes escalas de temperatura. Considere as fórmulas de conversão abaixo:

- De graus Celsius ($C$) para graus Fahrenheit ($F$): $F = (5/9) \times (C - 32)$
- De graus Celsius ($C$) para graus Kelvin ($K$): $K = C - 273.15$
- De graus Fahrenheit ($F$) para graus Celsius ($C$): $C = 32 + 9/5 \times K$
- De graus Kelvin ($K$) para graus Celsius ($C$): $C = K + 273.15$
- De graus Fahrenheit ($F$) para graus Kelvin ($K$): $K = (9/5) \times F - 459.67$
- De graus Kelvin ($K$) para graus Fahrenheit ($F$): $F = (5/9) \times (K + 459.63)$

**Exercício 5.12:** ⋆⋆

O que aconteceria se não colocássemos valores para as constantes na listagem 5.4 ? Explique.

**Exercício 5.13:** ⋆⋆

Escreva uma classe que contenha métodos estáticos para calcular as médias de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos `int` e `double`.

**Exercício 5.14:** ⋆⋆

Escreva uma classe que contenha métodos estáticos para calcular as somas de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos `int` e `double`.

**Exercício 5.15:** ⋆⋆⋆

Escreva uma classe que contenha os campos estáticos para representar os senos, co-senos e tangentes de ângulos em graus, de 0 a 330 graus, de 30 em 30 graus, de acordo com a tabela abaixo. Os campos podem ser nomeados *sinXXX*, *cosXXX* e *tanXXX* onde *XXX* é o ângulo. *Dica:* use a classe `ConstantesMatematicas` para obter as constantes que serão necessárias, e calcule as constantes *tanXXX* como sendo *sinXXX* divido por *cosXXX*.

| ângulo | seno | co-seno | ângulo | seno | co-seno |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 180 | 0 | 1 |
| 30 | $1/2$ | $1/2 \times \sqrt{3}$ | 210 | $-1/2$ | $-1/2 \times \sqrt{3}$ |
| 60 | $1/2 \times \sqrt{3}$ | $1/2$ | 240 | $-1/2 \times \sqrt{3}$ | $-1/2$ |
| 90 | 1 | 0 | 270 | $-1$ | 0 |
| 120 | $1/2 \times \sqrt{3}$ | $-1/2$ | 300 | $-1/2 \times \sqrt{3}$ | $1/2$ |
| 150 | $1/2$ | $-1/2 \times \sqrt{3}$ | 330 | $-1/2$ | $1/2 \times \sqrt{3}$ |



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 5.16:** ⋆⋆⋆

Escreva uma classe que contenha métodos estáticos para retornar o maior de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos `int` e `double`. *Dica:* os métodos podem ser chamados em cascata: para calcular o maior de três valores *a*, *b* e *c*, pode-se calcular o maior valor de *a* e *b*, e comparar este resultado com *c*. Veja também o exercício 5.17.

**Exercício 5.17:** ⋆⋆⋆

Escreva uma classe que contenha métodos estáticos para retornar o menor de dois, três, quatro e cinco valores, considerando que os argumentos e retorno dos métodos podem ser dos tipos `int` e `double`. *Dica:* os métodos podem ser chamados em cascata: para calcular o menor de três valores *a*, *b* e *c*, pode-se calcular o menor valor de *a* e *b*, e comparar este resultado com *c*. Veja também o exercício 5.16.

**Exercício 5.18:** ⋆⋆⋆

Escreva uma versão da classe `RegistroAcademico` que tenha o campo `númeroDeMatrícula` declarado como `static`, e que incremente o valor deste campo cada vez que uma instância da classe for criada. Escreva também uma aplicação que crie algumas instâncias da classe para demonstrar seu funcionamento. *Dica:* use a listagem 4.1 como base.

**Exercício 5.19:** ⋆⋆⋆

Escreva uma versão da classe `ContaBancariaSimplificada` que tenha um campo `númeroDaConta` declarado como `static`, e que incremente o valor deste campo cada vez que uma instância da classe for criada. Escreva também uma aplicação que crie algumas instâncias da classe para demonstrar seu funcionamento. *Dica:* use o modelo na listagem 1.2 como base.

**Exercício 5.20:** ⋆⋆⋆

Escreva uma classe `SerieLimitada`, que encapsula um valor inteiro sequencial como os usados em notas e séries de gravuras. Esta classe deve permite que um programa crie um número limitado de instâncias dela, cada uma numerada com um valor sequencial. O número total de instâncias é controlado pelo campo `máximoDeInstâncias`, declarado como `static final` e o de instâncias já criadas é controlado pelo campo `contador` declarado como `static`. Escreva também uma aplicação que crie algumas instâncias da classe para demonstrar seu funcionamento.

## 5.7 Exercícios complementares do capítulo 5

**Exercício 5.21:** ⋆

Podemos ter várias versões do método `main` em uma classe, usando a sobrecarga de métodos ? Explique.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 5.22:** ⋆

A distância média da Terra à Lua é de aproximadamente 382.000 quilômetros. Usando a classe `ConversaoDeUnidadesDeComprimento` (listagem 5.6), escreva um programa em Java que mostre qual é a distância média da Terra à Lua em milhas e pés. Escreva métodos adicionais para a classe `ConversaoDeUnidadesDeComprimento` se necessário.

**Exercício 5.23:** ⋆

A área de um campo de futebol é de 8.250 metros quadrados. Usando a classe `ConversaoDeUnidadesDeArea` (exercício 5.7) escreva um programa em Java que mostre qual é a área de um campo de futebol em pés quadrados, acres e centímetros quadrados. Escreva métodos adicionais para a classe `ConversaoDeUnidadesDeArea` se necessário.

**Exercício 5.24:** ⋆

O volume de uma piscina olímpica é de 1.890 metros cúbicos. Usando a classe `ConversaoDeUnidadesDeVolume` (exercício 5.8) escreva um programa em Java que mostre qual é o volume de uma piscina olímpica em litros, pés cúbicos e centímetros cúbicos. Escreva métodos adicionais para a classe `ConversaoDeUnidadesDeVolume` se necessário.

**Exercício 5.25:** ⋆

A velocidade da luz é de aproximadamente 1.080.000.000 quilômetros por hora. Usando a classe `ConversaoDeUnidadesDeVelocidade` (exercício 5.9) escreva um programa em Java que mostre qual é a velocidade da luz em milhas por hora, metros por segundo e pés por segundo. Escreva métodos adicionais para a classe `ConversaoDeUnidadesDeVelocidade` se necessário.

**Exercício 5.26:** ⋆

O tempo de gestação de um elefante indiano é de aproximadamente 624 dias. Usando a classe `ConversaoDeUnidadesDeTempo` (exercício 5.10) escreva um programa em Java que mostre qual é o tempo de gestação de um elefante indiano em dias, horas, minutos e segundos. Escreva métodos adicionais para a classe `ConversaoDeUnidadesDeVelocidade` se necessário.

**Exercício 5.27:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeTemperatura` (exercício 5.11), mostre quantos graus Kelvin e Fahrenheit correspondem a zero e cem graus Celsius.

# Capítulo 6

# Estruturas de Decisão e Controle - Condicionais

## 6.1 Introdução

Nos capítulos anteriores vimos que a criação de classes envolve a criação de métodos que processem dados destas classes ou passados como argumentos. O processamento de dados feito em uma classe ou aplicação muito frequentemente envolve decisões que devem ser feitas em relação a estes dados. A capacidade de tomar estas decisões de forma padronizada e programável (e, esperançosamente, correta) é uma das razões que faz o processamento de dados por computadores útil e interessante.

*decisões sobre processamento*

Mesmo modelos requerem, em sua maioria, decisões sobre processamento: *se* a lâmpada estiver queimada, não poderá ser ligada; *se* o saldo for menor que zero, o usuário não poderá retirar dinheiro da conta; *se* o ano for bissexto *e* o mês for fevereiro, *e* o dia for maior que 30, então a data estará incorreta; *se* o aluno for bolsista então deve ter um desconto de cinquenta por cento na sua mensalidade. Outros exemplos podem ser facilmente dados. Claramente, as classes baseadas nestes modelos deverão usar estruturas de decisão para implementar os mecanismos de processamento de dados.

*estruturas de decisão*

Neste capítulo veremos todos os mecanismos de Java para implementar processos de decisão. Veremos também os operadores lógicos que controlam estes mecanismos.

## 6.2 Operadores lógicos em Java

*operadores lógicos*

Todas as estruturas de controle de fluxo de um programa ou método são baseadas em *operadores lógicos*: estruturas de decisão executarão parte do código *se* uma condição ocorrer ou não, e estruturas de repetição repetirão trechos de código *até que* uma condição seja cumprida ou *enquanto* uma condição for válida.

*operadores de comparação*

Operadores lógicos para controle destas estruturas são operadores de comparação, que comparam dois valores para ver se são iguais, diferentes, ou se um é maior, menor, maior ou igual ou menor ou igual a outro, retornando um valor booleano igual a `true` ou `false` dependendo do resultado. Esses operadores tem a forma genérica `valor operador valor`, onde os valores podem ser campos, variáveis, constantes ou resultados de operações e o operador pode ser um dos listados a seguir:

- `==` retorna o valor booleano `true` se os dois valores forem **exatamente iguais** e `false` *operador ==*



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

se forem **diferentes**.

*operador !=*
- `!=` retorna o valor booleano `true` se os dois valores forem **diferentes** e `false` se forem **exatamente iguais** (inverso do caso anterior).

*operador >*
- `>` retorna o valor booleano `true` se o valor à esquerda do sinal for **maior** do que o valor à direita do sinal, e `false` caso for **menor** ou **igual**.

*operador <*
- `<` retorna o valor booleano `true` se o valor à esquerda do sinal for **menor** do que o valor à direita do sinal, e `false` caso for **maior** ou **igual**.

*operador >=*
- `>=` retorna o valor booleano `true` se o valor à esquerda do sinal for **maior** ou **igual** do que o valor à direita do sinal, e `false` se for **menor**.

*operador <=*
- `<=` retorna o valor booleano `true` se o valor à esquerda do sinal for **menor** ou **igual** do que o valor à direita do sinal, e `false` se for **maior**.

*comparação de referências*

Os operadores acima podem ser usados para comparar valores de tipos nativos numéricos (inclusive `char`). Referências às instâncias de classes podem ser comparadas com os operadores `==` e `!=`, mas os resultados geralmente são inesperados e incorretos: duas referências diferentes a instâncias de classes contendo os mesmos dados serão diferentes quando comparadas. Se duas referências que apontem para a mesma instância forem comparadas com o operador `==`, o resultado será `true`.

Outros operadores de comparação não podem ser usados com instâncias de classes (inclusive da classe String) - estas devem ser comparadas com métodos específicos. Para a classe `String` estes métodos serão vistos no capítulo 12. Para outras classes, inclusive para classes definidas pelo programador, devemos ter métodos que aceitem outra instância da classe e façam a comparação, retornando o valor booleano adequado.

*comparação de valores de ponto flutuante*

Cuidado especial deve ser tomado quando valores de ponto flutuante (`double` e `float`) forem comparados usando os operadores `==` e `!=`: como os valores de ponto flutuante são representados de maneira aproximada, comparações absolutas podem dar resultados diferentes dos esperados. Um exemplo clássico é o da expressão `3/97-(2/97+1/97) == 0.0`, que será igual a `false` se os valores usados forem do tipo `float`, e igual a `true` se os valores forem do tipo `double` (por causa da capacidade dos `double`s de representarem os valores com maior precisão). Vale a pena notar que expressões deste tipo com denominadores diferentes podem resultar em `false` até mesmo para valores do tipo `double`: em suma, comparações absolutas com valores de ponto flutuante devem ser evitadas, se possível.

*comparação de valores booleanos*

Valores booleanos somente podem ser comparados com os operadores `==` e `!=`. Se a comparação for feita entre campos ou variáveis do tipo `boolean` e constantes booleanas, podemos escrevê-las abreviadamente: a comparação `(campo == true)` pode ser escrita simplesmente como `(campo)`, uma vez que a expressão somente será verdadeira se `campo` for igual a `true`. Similarmente, podemos escrever expressões do tipo `(variável != true)` ou `(variável == false)` como `(!variável)`.

Valores do tipo `boolean` (constantes, variáveis ou campos) e resultados de operações ou métodos que retornem valores booleanos podem ser combinados entre si, na forma `valor operador valor`, onde os valores devem ser do tipo booleano e os operadores são os listados a seguir:

*operador && E lógico*
- `&&` retorna o valor booleano `true` se e somente se os dois valores (à esquerda e à direita do operador) valerem `true`. Este é o operador booleano **AND** (ou **E** lógico).

*operador || OU lógico*
- `||` retorna o valor booleano `true` se ao menos um dos dois valores (à esquerda e à direita do sinal) valerem `true`. Este é o operador booleano **OR** (ou **OU** lógico). Note que o sinal usado é a barra vertical e **não** o sinal de exclamação (`!`).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Mais de duas expressões, constantes, resultados de métodos, etc. podem ser compostos em uma operação com resultado booleano usando o **E** lógico ou **OU** lógico, na forma `valor operador valor operador valor operador valor...`. Para manter a clareza, é aconselhável agrupar estas operações entre parênteses.

*operador* `!` `NÃO` *lógico*

Um outro operador booleano é o **NOT** (**NÃO** lógico ou negação), que transforma um valor booleano, constante, resultado de chamada a método, etc. de `true` para `false` e vice-versa. O operador em Java que faz a negação lógica é o sinal de exclamação (`!`), e que deve preceder um valor ou operação cujo resultado seja booleano.

A classe `Comparavel`[1] exemplifica o uso dos operadores de comparação, encapsulando um valor que será comparado, através de métodos, com outros valores passados como argumentos. A classe `Comparavel` é mostrada na listagem 6.1.

Listagem 6.1: A classe `Comparavel`, que encapsula um valor e contém métodos para compará-lo com outros.

```
1  /**
2   * A classe Comparavel, que encapsula um valor do tipo double e tem métodos que
3   * permitem comparar este valor com outros passados como argumentos de diversas
4   * formas, exemplificando o uso dos compadores de Java.
5   */
6  class Comparavel // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10  */
11  private double valor; // o valor encapsulado, que será comparado com outros
12
13  /**
14   * O construtor para a classe Comparavel, que recebe um argumento do tipo double
15   * para inicializar o único campo da classe.
16   * @param v o valor a ser encapsulado
17   */
18  Comparavel(double v)
19    {
20    valor = v;
21    } // fim do construtor com um argumento
22
23  /**
24   * O método toString possibilita a impressão de instâncias desta classe.
25   * @return o valor encapsulado convertido para uma String
26   */
27  public String toString()
28    {
29    return ""+valor; // converte para String via concatenação
30    } // fim do método toString
31
32  /**
33   * O método éIgualA retorna o valor booleano true se o valor encapsulado nesta classe
34   * for igual ao passado como argumento.
35   * @param v1 o valor a ser comparado para igualdade
36   * @return true se os valores forem iguais, false se forem diferentes
37   */
38  public boolean éIgualA(double v1)
39    {
40    if (valor == v1) return true; // comparação de igualdade de valores
41    else return false;
42    } // fim do método éIgualA
```

[1] Aparentemente esta classe é uma complicação desnecessária, mas pode ser usada como base para a criação de classes mais complexas e que reduzirão o trabalho do programador usuário. Veja a lista de exercícios deste capítulo para sugestões de melhorias nesta classe.

```
43
44  /**
45   * O método éIgualA retorna o valor booleano true se o valor encapsulado nesta classe
46   * for igual aos passados como argumento.
47   * @param v1 um valor a ser comparado para igualdade
48   * @param v2 outro valor a ser comparado para igualdade
49   * @return true se os valores forem iguais, false se forem diferentes
50   */
51   public boolean éIgualA(double v1,double v2)
52     {
53     if ((valor == v1) && (valor == v2)) // ambos os valores são iguais ao encapsulado
54       return true;
55     else return false;
56     } // fim do método éIgualA
57
58  /**
59   * O método éMaiorQue retorna o valor booleano true se o valor encapsulado nesta
60   * classe for maior que o passado como argumento.
61   * @param v1 o valor a ser comparado
62   * @return true se o valor encapsulado for maior que o passado como argumento,
63   *         false caso contrário
64   */
65   public boolean éMaiorQue(double v1)
66     {
67     if (valor > v1) return true; // o valor encapsulado é maior que o argumento
68     else return false;
69     } // fim do método éMaiorQue
70
71  /**
72   * O método éMaiorQue retorna o valor booleano true se o valor encapsulado nesta
73   * classe for maior que os passados como argumentos.
74   * @param v1 um valor a ser comparado
75   * @param v2 outro valor a ser comparado
76   * @return true se o valor encapsulado for maior que os passados como argumentos,
77   *         false caso contrário
78   */
79   public boolean éMaiorQue(double v1,double v2)
80     {
81     if ((valor > v1) && (valor > v2)) // o valor encapsulado é maior que os argumentos
82       return true;
83     else return false;
84     } // fim do método éMaiorQue
85
86  /**
87   * O método éMenorQue retorna o valor booleano true se o valor encapsulado nesta
88   * classe for menor que o passado como argumento.
89   * @param v1 o valor a ser comparado
90   * @return true se o valor encapsulado for menor que o passado como argumento,
91   *         false caso contrário
92   */
93   public boolean éMenorQue(double v1)
94     {
95     if (valor < v1) return true; // o valor encapsulado é menor que o argumento
96     else return false;
97     } // fim do método éMenorQue
98
99  /**
100  * O método éMenorQue retorna o valor booleano true se o valor encapsulado nesta
101  * classe for menor que os passados como argumentos.
102  * @param v1 um valor a ser comparado
103  * @param v2 outro valor a ser comparado
104  * @return true se o valor encapsulado for menor que os passados como argumentos,
105  *         false caso contrário
106  */
107  public boolean éMenorQue(double v1,double v2)
108    {
109    if ((valor < v1) && (valor < v2)) // o valor encapsulado é menor que os argumentos
110      return true;
111    else return false;
```

```
112     } // fim do método éMenorQue
113
114   } // fim da classe Comparavel
```

Cada um dos métodos da classe `Comparavel` (listagem 6.1) é baseado em um operador de comparação da linguagem. Todos retornam um valor do tipo `boolean` indicando se a operação de comparação resultou em verdadeiro ou falso. A classe `Comparavel` esconde do programador usuário os símbolos dos operadores de comparação, substituindo-os por métodos com nomes descritivos, também permitindo comparações com mais de um valor simultaneamente.

A classe `DemoComparavel`, mostrada na listagem 6.2, demonstra o uso dos métodos de uma instância da classe `Comparavel`, assim como a combinação dos métodos que retornam um valor do tipo `boolean` com os operadores de combinação.

Listagem 6.2: A classe `DemoComparavel`, que demonstra usos de instâncias e métodos da classe `Comparavel`.

```
1  /**
2   * A classe DemoComparavel, que demonstra o uso de instâncias da classe Comparavel.
3   */
4  class DemoComparavel // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém a declaração
8    * de uma instância da classe Comparavel, e demonstra seu uso.
9    * @param argumentos os argumentos que podem ser passados para o método via linha
10   *        de comando, mas que neste caso serão ignorados.
11   */
12    public static void main(String[] argumentos)
13      {
14      // Criamos uma instância da classe Comparavel que encapsula o valor 123
15      Comparavel meuValor = new Comparavel(123);
16      // Criamos alguns valores constantes para usar com os métodos
17      double valor1 = 120;
18      double valor2 = 122;
19      // Executamos os métodos de comparação da classe Comparavel - primeiro a igualdade
20      if (meuValor.éIgualA(valor1))
21        System.out.println("Os valores "+meuValor+" e "+valor1+" são iguais.");
22      if (meuValor.éIgualA(valor1,valor2))
23        System.out.println("Os valores "+meuValor+", "+valor2+" e "+valor1+
24                           " são iguais.");
25      // Comparamos para ver se o encapsulado é maior que os passados
26      // (somente os comandos associados a estes "if"s serão executados)
27      if (meuValor.éMaiorQue(valor1))
28        System.out.println("O valor "+meuValor+" é maior que o valor "+valor1);
29      if (meuValor.éMaiorQue(valor1,valor2))
30        System.out.println("O valor "+meuValor+" é maior que os valores "+valor1+" e "+
31                           valor2);
32      // Comparamos para ver se o encapsulado é menor que os passados
33      if (meuValor.éMenorQue(valor1))
34        System.out.println("O valor "+meuValor+" é menor que o valor "+valor1);
35      if (meuValor.éMenorQue(valor1,valor2))
36        System.out.println("O valor "+meuValor+" é menor que os valores "+valor1+" e "+
37                           valor2);
38      } // fim do método main
39
40   } // fim da classe DemoComparavel
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

## 6.3 Estruturas de decisão e execução seletiva

A maior utilidade dos operadores de comparação e de combinações de valores booleanos é seu uso para tomar decisões em programas. As decisões basicamente indicam quais trechos de código serão executados e quais não serão. Nesta seção veremos duas estruturas de decisão e execução seletiva, baseadas na avaliação de expressões.

### 6.3.1 As instruções `if-else`

`if-else`

As instruções `if-else` permitem que um comando ou bloco de comandos seja executado dependendo do resultado de uma comparação, combinações lógicas ou de qualquer método que retorne um valor do tipo `boolean`. A estrutura básica destas instruções é `if (valor-booleano) comando-ou-bloco else comando-ou-bloco` onde `comando-ou-bloco` pode ser um único comando (obrigatoriamente terminado por ponto-e-vírgula) ou bloco de comandos (obrigatoriamente delimitados por `{` e `}`).

Se a expressão associada à palavra-chave `if` for igual a `true`, o comando ou bloco de comandos associado ao `if` será executado, caso contrário o comando ou bloco de comandos associado ao `else` será executado. Em outras palavras, a execução dos blocos de comandos é *mutuamente exclusiva*: somente um dos dois será executado.

A instrução `else` e o bloco associado a ela é opcional, caso não seja usada e a expressão avaliada pelo `if` for falsa, o programa ou método continuará seguindo seu fluxo normal.

`if-else` *aninhados*

Instruções `if-else` podem ser *aninhadas*, isto é, os blocos de execução associados ao `if` ou ao `else` pode conter outras instruções `if-else` e respectivos blocos.

A listagem 6.3 mostra uma classe com o método `main` que demonstra o uso das instruções `if-else`, com um exemplo de aninhamento. Esta classe demonstra a implementação em Java de um algoritmo simples para cálculo do preço de entrada de cinema, em pseudo-código:

```
1  Se o dia da semana é terça, quarta ou quinta
2    início
3      preço = 4 reais
4    fim
5  senão
6    início
7      se a idade da pessoa é menor que 14 anos ou maior que 65 anos
8        início
9          preço = 4 reais
10       fim
11     senão
12       início
13         preço = 8 reais
14       fim
15   fim
```

O cálculo do preço é feito pelo método estático `calculaPreço`, chamado a partir do método `main` com as variáveis necessárias para o cálculo passadas como argumentos.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem 6.3: A classe `EntradaDeCinema`, que calcula o preço de uma entrada de cinema.

```
/**
 * A classe EntradaDeCinema, que calcula o preço de uma entrada de cinema usando um
 * método estático.
 */
class EntradaDeCinema // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método pergunta alguns
  * dados ao usuário e os passa como argumentos para um método que calcula e retorna
  * o preço a ser pago por uma entrada de cinema.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Algumas variáveis que servirão para cálculo do preço da entrada
    short idade;
    byte diaDaSemana;
    // Perguntamos os dados ao operador do programa (usuário)
    System.out.print("Entre a idade do comprador:");
    idade = Keyboard.readShort();
    System.out.print("Entre o dia da semana (1=domingo, 2=segunda,..., 7=sábado):");
    diaDaSemana = Keyboard.readByte();
    // Chama o método para calcular o preço
    float preço = calculaPreço(idade,diaDaSemana);
    // Imprime o resultado
    System.out.println("O preço da entrada é "+preço+" reais");
    } // fim do método main

 /**
  * O método calculaPreço calcula o preço da entrada de cinema usando o seguinte
  * algoritmo:
  * - Se a pessoa tem menos de 14 anos ou mais de 65 anos, paga meia entrada
  *   (4 reais), caso contrário paga oito reais
  * - Se o dia da semana for terça, quarta ou quinta (3, 4 ou 5), todos podem pagar
  *   meia entrada, independente da idade.
  * @param idade a idade do comprador
  * @param diaDaSemana o dia da semana, entre 1 (domingo) e 7 (sábado).
  * @return o preço da entrada do cinema.
  */
  static float calculaPreço(short idade,byte diaDaSemana)
    {
    float preço; // armazenará o preço calculado
    // Se o dia da semana é entre 3 e 5, todo mundo paga meia
    if ((diaDaSemana >= 3) && (diaDaSemana <= 5))
      preço = 4;
    else // então é outro dia da semana, vamos ver de acordo com a idade
      {
      if ((idade <= 14) || (idade >= 65))
        preço = 4;
      else
        preço = 8;
      } // fim do else do dia da semana
    return preço;
    } // fim do método calculaPreço

  } // fim da classe EntradaDeCinema
```

*cuidados com `if-else` aninhados*

Se o aninhamento de `if`s e `else`s for feito, cuidado especial deverá ser tomado para garantir que um `else` estará associado ao `if` correto, através do uso de blocos. Como o `else` é sempre associado ao `if` imediatamente anterior, se o bloco de comandos associado à este `if` conter outro `if`, o uso dos delimitadores `{` e `}` será obrigatório. Como exemplo deste problema, considere o seguinte trecho de código:

```
1  if (condição1)
2    if (condição2)
3      System.out.println("Condições 1 e 2 são verdadeiras");
4  else
5    System.out.println("Condição 1 é falsa");
```

No trecho de código acima, o `else` da linha 4 será executado caso o `if` mais próximo, ou seja, o da linha 2, não for executado - o resultado impresso pelo `else` não estaria correto se `condição1` fosse verdadeira e `condição2` fosse falsa. O trecho de código a seguir mostra como o `else` pode ser associado ao primeiro `if`.

```
1  if (condição1)
2    {
3    if (condição2)
4      System.out.println("Condições 1 e 2 são verdadeiras");
5    }
6  else
7    System.out.println("Condição 1 é falsa");
```

No trecho de código acima, o `else` da linha 6 será executado caso o `if` da linha 1 não seja executado.

### `if-else` *em cascata*

Blocos de `if-else`s também podem ser dispostos em *cascata* ou *encadeados*, de forma que se uma condição não for satisfeita em um `if`, a seguinte será avaliada e assim em diante. A classe `DataIf`, mostrada na listagem 6.4, mostra no método `mostraData` como vários `if`s podem ser encadeados de forma que uma única das condições seja executada.

Listagem 6.4: A classe `DataIf`, com um método que demonstra a instrução `if` em cascata.

```
1  /**
2   * A classe DataIf, que contém campos e métodos que permitem a manipulação de datas.
3   * Esta versão da classe não tem métodos que seriam úteis para seu uso, somente
4   * contém o construtor (que não verifica a validade dos dados) e o método toString
5   * que demonstra a instrução if usada em cascata (objetivo desta classe).
6   */
7  class DataIf // declaração da classe
8    {
9   /**
10   * Declaração dos campos da classe
11   */
12   private byte dia,mês; // dia e mês são representados por bytes
13   private short ano; // ano é representado por um short
14
15  /**
16   * O construtor da classe DataIf recebe argumentos para inicializar os campos da
17   * classe. Este construtor não verifica a validade da data, veja o comentário no
18   * início da classe para maiores informações.
19   * @param d o argumento correspondente ao método dia
20   * @param m o argumento correspondente ao método mês
21   * @param a o argumento correspondente ao método ano
22   */
23   DataIf(byte d,byte m,short a)
24     {
25     dia = d; mês = m; ano = a;
26     } // fim do construtor
27
28  /**
29   * O método toString retorna uma String contendo os valores dos campos formatados
30   * (ou seja, a data com o mês por extenso)
31   * @return uma String com a data formatada
32   */
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
33   public String toString()
34     {
35     String resultado = ""+dia;
36     resultado += " de ";
37     if (mês == 1) resultado += "Janeiro";        // Verifica qual é o mês e
38     else if (mês == 2) resultado += "Fevereiro"; // imprime o nome
39     else if (mês == 3) resultado += "Março";
40     else if (mês == 4) resultado += "Abril";
41     else if (mês == 5) resultado += "Maio";
42     else if (mês == 6) resultado += "Junho";
43     else if (mês == 7) resultado += "Julho";
44     else if (mês == 8) resultado += "Agosto";
45     else if (mês == 9) resultado += "Setembro";
46     else if (mês ==10) resultado += "Outubro";
47     else if (mês ==11) resultado += "Novembro";
48     else if (mês ==12) resultado += "Dezembro";
49     resultado += " de "+ano;
50     return resultado;
51     }  // fim do método toString
52
53   } // fim da classe DataIf
```

Um problema inerente à construção de longos `if`s encadeados é que eles podem se tornar ineficientes: no método `mostraData` da classe `DataIf` na listagem acima, caso o campo `mês` valha 12, onze `if`s deverão ser executados em seqüência, o que não é eficiente. Uma solução melhor será apresentada na seção 6.3.3.

*problemas com `if-else` em cascata*

### 6.3.2 O operador condicional `?`

Nos casos em que o objetivo de uma avaliação de expressão pela instrução `if` é simplesmente o de determinar que valor será atribuído a uma variável, o operador condicional `?` poderá ser usado. A forma básica deste operador é `valor = (expressão ? valor_se_verdadeiro : valor_se_falso);`, que poderia ser escrito, usando `if`s, como `if (expressão) valor = valor_se_verdadeiro; else valor = valor_se_falso;`. A classe `ComparaSimples`, na listagem 6.5, demonstra o uso do operador condicional `?`.

*operador condicional `?`*

Listagem 6.5: A classe `ComparaSimples`, que demonstra o uso do operador condicional `?`.

```
1  /**
2   * A classe ComparaSimples, que contém métodos estáticos para a comparação de dois
3   * valores do tipo inteiro. Esta classe demonstra o uso do operador condicional ?.
4   */
5  class ComparaSimples // declaração da classe
6    {
7   /**
8    * O método qualÉMaior retorna o maior dos dois valores passados como argumentos.
9    * @param p1 o primeiro dos dois valores
10   * @param p2 o segundo dos dois valores
11   * @return o maior dos dois valores passados como argumentos
12   */
13   public static int qualÉMaior(int p1,int p2)
14     {
15     int valor = (p1 > p2 ? p1 : p2); // se p1 > p2, valor recebe p1, senão recebe p2
16     return valor;
17     } // fim do método qualÉMaior
18
19  /**
20   * O método qualÉMenor retorna o menor dos dois valores passados como argumentos.
21   * @param p1 o primeiro dos dois valores
22   * @param p2 o segundo dos dois valores
23   * @return o menor dos dois valores passados como argumentos
24   */
25   public static int qualÉMenor(int p1,int p2)
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
26      {
27      // Não é necessário declarar uma variável temporária
28      return (p1 < p2 ? p1 : p2); // se p1 > p2, retorna p1, senão retorna p2
29      } // fim do método qualÉMenor
30
31    } // fim da classe ComparaSimples
```

Vale a pena notar que apesar da forma de uso do operador condicional `?` ser mais compacta, em geral o uso de `if`s torna o programa mais claro.

### 6.3.3 A instrução `switch`

`switch`

Quando for necessária a avaliação de um valor de um tipo inteiro para execução de um trecho de código para cada um dos valores possíveis, como foi o caso do método `mostraData` da classe `DataIf` (listagem 6.4), existe uma instrução que faz com que o código fique mais claro e eficiente: a instrução `switch`.

A instrução `switch` permite que um valor ou expressão que retorne um valor do tipo `int` ou de menor precisão (`byte`, `short` ou `char` mas não `long`) seja avaliado, e dependendo do valor, o fluxo do programa ou rotina será modificado para uma posição específica, emulando de forma diferente o funcionamento de blocos `if-else` em cascata.

`case`

A instrução `case` deve ser seguida de um bloco contendo ao menos uma palavra-chave `case` seguida de um valor constante de tipo compatível com o que é usado para avaliação pela instrução `switch`, seguida de comandos que devem ser executados caso a expressão ou valor avaliada pelo `switch` seja igual à constante do `case`.

As instruções `switch` e `case` são demonstradas em dois métodos da classe `DataSwitch`, mostrada na listagem 6.6.

Listagem 6.6: A classe `DataSwitch`, com um método que demonstra a instrução `switch`.

```
1  /**
2   * A classe DataSwitch, que contém campos e métodos que permitem a manipulação de
3   * datas. Esta versão da classe não tem métodos que seriam úteis para seu uso, somente
4   * contém o construtor (que não verifica a validade dos dados) e dois métodos que
5   * demonstram a instrução switch (objetivo desta classe).
6   */
7  class DataSwitch // declaração da classe
8    {
9   /**
10   * Declaração dos campos da classe
11   */
12   private byte dia,mês; // dia e mês são representados por bytes
13   private short ano; // ano é representado por um short
14
15  /**
16   * O construtor da classe DataSwitch recebe argumentos para inicializar os campos da
17   * classe. Este construtor não verifica a validade da data, veja o comentário no
18   * início da classe para maiores informações.
19   * @param d o argumento correspondente ao método dia
20   * @param m o argumento correspondente ao método mês
21   * @param a o argumento correspondente ao método ano
22   */
23   DataSwitch(byte d,byte m,short a)
24     {
25     dia = d; mês = m; ano = a;
26     } // fim do construtor
27
28  /**
```

```
29   * O método toString retorna uma String contendo os valores dos campos formatados
30   * (ou seja, a data com o mês por extenso)
31   * @return uma String com a data formatada
32   */
33   public String toString()
34     {
35     String resultado = ""+dia;
36     resultado += " de ";
37     switch(mês) // dependendo do valor do campo mês, concatena um dos nomes abaixo.
38       {
39       case  1: resultado += "Janeiro"; break;
40       case  2: resultado += "Fevereiro"; break;
41       case  3: resultado += "Março"; break;
42       case  4: resultado += "Abril"; break;
43       case  5: resultado += "Maio"; break;
44       case  6: resultado += "Junho"; break;
45       case  7: resultado += "Julho"; break;
46       case  8: resultado += "Agosto"; break;
47       case  9: resultado += "Setembro"; break;
48       case 10: resultado += "Outubro"; break;
49       case 11: resultado += "Novembro"; break;
50       case 12: resultado += "Dezembro"; break;
51       }
52     resultado += " de "+ano;
53     return resultado;
54     }  // fim do método toString
55
56  /**
57   * O método diasNoMês retorna o número de dias que existe no mês encapsulado nesta
58   * classe. O algoritmo simplifica, assumindo que o ano não é bissexto.
59   * @return o número de dias no mês da data encapsulada
60   */
61   public byte diasNoMês()
62     {
63     byte númeroDeDias;
64     switch(mês) // dependendo do valor do campo mês, armazenamos um número de dias
65       {
66       case  2: númeroDeDias = 28; break;
67       case  4:
68       case  6:
69       case  9:
70       case 11: númeroDeDias = 30; break;
71       default: númeroDeDias = 31; break;
72       }
73     return númeroDeDias;
74     }  // fim do método diasNoMês
75
76   } // fim da classe DataSwitch
```

Alguns comentários sobre a listagem 6.6 e sobre a estrutura `switch/case` são mostrados a seguir.

- No corpo do método `mostraData` da classe `DataSwitch` vemos o primeiro exemplo de uso da instrução `switch`, relativamente simples: o valor do mês (que é do tipo `byte` e esperamos estar entre 1 e 12) é avaliado e o fluxo do método passa da instrução `switch` para a `case` com a constante igual ao valor avaliado. Desta forma, se o mês encapsulado for sete, o fluxo do programa passará da linha 38 para a 46.

`break` *em* `switch` *mecanismo de funcionamento do* `switch`

- Todas as linhas após as instruções `case` do método `mostraData` da classe `DataSwitch` são terminadas com um comando `break`. Isto é feito porque a instrução `switch` transfere o fluxo do método em execução diretamente para o `case` correspondente (que é o ponto de entrada do bloco `switch`), mas não controla o fluxo do programa ou método depois que os comandos associados com o `case` são executados - o comando `break` faz com que o fluxo de execução seja transferido para a primeira linha depois do bloco `case`, no caso deste método, a linha 53. Se os comandos `break` não fossem usados, o programa ou método



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

continuaria a executar os outros comandos diretamente: no caso do método `mostraData`, imprimindo todos os meses.

- O valor associado a cada `case` **deve** ser uma constante do tipo inteiro ou compatível (`char`, `byte`, etc.). Constantes do tipo `long` podem, em princípio, ser usadas, mas como o compilador Java insistirá na conversão explícita destas constantes para o tipo `int`, não existe razão para usar valores do tipo `long` nem nos `case` nem como argumento para o `switch`. Não é possível avaliar valores do tipo `boolean`, valores de ponto flutuante (`float` e `double`) ou instâncias de qualquer classe.
- Não é necessário que os valores associados aos `case`s sejam contíguos ou mesmo que estejam em ordem, mas não é possível ter duas instruções `case` com valores associados iguais.
- Não é necessário incluir os comandos associados a um `case` entre chaves (`{` e `}`), mas fazê-lo contribui para a clareza da classe ou programa, em especial quando existirem vários comandos associados ao `case`.
- Dentro do bloco de uma instrução `switch` podemos ter outros blocos `switch`, `if`, laços de repetição, chamada a métodos, etc.
- No corpo do método `diasNoMês` vemos o segundo exemplo de uso da instrução `switch` com vários `case`s. Neste exemplo, vemos que nem sempre é necessário usar o comando `break` para interromper o processamento, em especial, quando queremos que para vários valores diferentes avaliados pelo `switch` o mesmo bloco de comandos seja executado, o que ocorre neste método: caso o valor do campo `mês` seja 4, 6, 9 ou 11, a variável `númeroDeDias` receberá o valor 30.

`default`

- No corpo do método `diasNoMês` vemos também o uso da palavra-chave `default`, que será o próximo ponto de execução do bloco `switch` caso nenhum dos valores associados aos `case`s seja igual ao avaliado pela instrução `switch`. Em outras palavras, se o fluxo do programa ou método não for transferido para nenhum dos `case`, será transferido para o `default`, se este existir no do bloco `switch`. Se não existir uma entrada `default`, nada será feito, e o programa ou método continuará a execução depois do bloco `switch`.
- O último ponto de entrada em um bloco `switch` não precisa ter o comando `break`, pois após a sua execução, o bloco `switch` será terminado de qualquer forma.
- Curiosamente, o bloco associado a palavra-chave `default` não precisa ser a última do bloco do `switch`, mas por clareza é aconselhável que seja.

*comparação do `switch` com `if-else`*

As vantagens do uso do `switch` quando comparado com blocos encadeados de `if`s e `else`s é que o `switch` é mais rápido, dispensando a execução de várias avaliações, e em geral, o programa fica mais claro e simples de ser lido. A desvantagem é que as regras de utilização do bloco `switch` são rigorosas, não permitindo comparações diferentes da igualdade: por exemplo, não podemos fazer algo como `case > 100` ou outras comparações que não sejam a igualdade.

## 6.4 Exercícios do capítulo 6

**Exercício 6.1:** ⋆

Escreva para a classe `Comparavel` (listagem 6.1) o método `éIgualAQualquerUmDe` que aceite dois valores como argumentos e retorne `true` se o valor encapsulado for igual a qualquer um dos passados como argumentos.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 6.2:** ⋆

Escreva versões do método `éIgualAQualquerUmDe` (veja o exercício 6.1) que aceitem três, quatro e cinco valores do tipo `double` como argumentos, e retorne `true` se o valor encapsulado for igual a qualquer um dos valores passados como argumentos.

**Exercício 6.3:** ⋆

Escreva para a classe `Comparavel` (listagem 6.1) o método `éMaiorOuIgual` que receba um valor do tipo `double` como argumento e retorne `true` se o valor encapsulado for maior ou igual ao passado como argumento. *Dica:* este problema pode também ser resolvido usando-se os métodos `éIgualA` e `éMaiorQue`, já existentes na classe, e a operação booleana "ou".

**Exercício 6.4:** ⋆

Escreva para a classe `Comparavel` (listagem 6.1) o método `éMenorOuIgual` que receba um valor do tipo `double` como argumento e retorne `true` se o valor encapsulado for menor ou igual ao passado como argumento. *Dica:* este problema pode também ser resolvido usando-se os métodos `éIgualA` e `éMenorQue`, já existentes na classe, e a operação booleana "ou".

**Exercício 6.5:** ⋆

Escreva para a classe `Comparavel` (listagem 6.1) o método `éDiferenteDe` que receba um valor do tipo `double` como argumento e retorne `true` se o valor encapsulado for diferente do passado como argumento. *Dica:* este problema pode também ser resolvido usando-se os métodos `éMaiorQue` e `éMenorQue`, já existentes na classe, e a operação booleana "e". Outra forma de resolver este exercício é usar o método `éIgualA` e a operação booleana de negação.

**Exercício 6.6:** ⋆

Escreva versões do método `éDiferenteDe` (veja o exercício 6.5) que aceitem três, quatro e cinco valores do tipo `double` como argumentos, e retorne `true` se o valor encapsulado for diferente de todos os valores passados como argumentos.

**Exercício 6.7:** ⋆

O método `calculaPreço` na classe `EntradaDeCinema` (listagem 6.3) verifica primeiro se o dia da semana é dia de desconto, para depois verificar a idade do cliente. Modifique este método para que primeiro a idade seja verificada, para depois verificar o dia da semana, de forma que o resultado final seja o mesmo.

**Exercício 6.8:** ⋆

O método `mudaDireção` da classe `RoboSimples` (listagem 4.7) não verifica se a direção passada como argumento é uma das direções válidas (`'N'`, `'S'`, `'E'` ou `'O'`). Modifique o método de forma que se um caracter diferente dos aceitos como direções válidas for passado, o método considere a direção como sendo `'N'`.

**Exercício 6.9:** ⋆

Modifique o método `calculaPreço` da classe `EntradaDeCinema` (listagem 6.3) para que este também considere que horas são, e retorne o preço de meia entrada caso sejam antes das quatro horas.

**Exercício 6.10:** ⋆

O que aconteceria se todos os `else` fossem retirados do método `mostraData` da classe `DataIf` (listagem 6.4) ? Existe alguma vantagem ou desvantagem em fazer isto ?

**Exercício 6.11:** ⋆

Modifique o método `diasNoMês` da classe `DataSwitch` (listagem 6.6) para que ele use comandos `if` ao invés de `switch`.

**Exercício 6.12:** ⋆

Ao final do laço que controla as tentativas de acerto no método `tenta` da classe `JogoDeAdivinhacao` (listagem 7.4) (linhas 53 a 56), o número de tentativas é avaliado para verificar se o usuário acertou ou não o número secreto. Que outra condição poderia ser usada para esta verificação ? Modifique o método `tenta` para usar esta outra condição.

**Exercício 6.13:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class ErroNoSwitch
2    {
3    public static void main(String[] argumentos)
4      {
5      System.out.print("Entre um valor entre zero e dez");
6      byte valor = Keyboard.readByte();
7      switch(valor)
8        {
9        case     0:
10       case     1: System.out.println("Valor baixo..."); break;
11       case     9:
12       case    10: System.out.println("Valor alto..."); break;
13       case 99999: System.out.println("Valor exageradamente..."); break;
14       default   : System.out.println("Valor normal."); break;
15       }
16     }
17   } // fim da classe
```

**Exercício 6.14:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class MultiplaEscolha
2    {
3    public static void main(String[] argumentos)
4      {
5      System.out.println("Escolha a opção correta:");
6      System.out.println("P - Quantos lados tem um círculo ?");
7      System.out.println("1 - Nenhum.");
8      System.out.println("2 - Dois.");
9      System.out.println("3 - Infinitos.");
10     System.out.println("4 - Nenhuma das respostas acima.");
11
12     byte resposta = Keyboard.readByte();
13     switch(resposta)
14       {
15       default   : System.out.println("Você não forneceu resposta !"); break;
16       case     4: System.out.println("Correto."); break;
17       case     1:
18       case     2:
19       case     3:
20       case     4: System.out.println("Não estou bem certo..."); break;
21       default   : System.out.println("Errado."); break;
22       }
23     }
24   } // fim da classe
```

**Exercício 6.15:** ⋆

Explique e exemplifique o que aconteceria com o método `diasNoMês` da classe `DataSwitch` (listagem 6.6) se os comandos `break` fossem retirados das instruções `case` do método.

**Exercício 6.16:** ⋆

Modifique o método `calculaMensalidade` na classe `RegistroAcademicoSemConstrutor` (listagem 4.1) para que este use a instrução `switch` ao invés de um bloco de `if-else`s.

**Exercício 6.17:** ⋆⋆

Escreva métodos adicionais de comparação para a classe `Comparavel` (listagem 6.1) usando sobrecarga de métodos, que recebam como argumentos outras instâncias da própria classe `Comparavel`.

**Exercício 6.18:** ⋆⋆

Escreva, para a classe `Ponto2D` (listagem 3.2) os métodos `estáAcimaDe`, `estáAbaixoDe`, `estáÀEsquerdaDe` e `estáÀDireitaDe` que recebem como argumento uma outra instância de `Ponto2D` e retornam `true` se o ponto encapsulado estiver, respectivamente, acima, abaixo, à esquerda e à direita do ponto passado como argumento. Veja também a figura 3.3.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 6.19:** ⋆⋆

Escreva uma classe que encapsule um menu de lanchonete com um método estático `imprimeMenu` para imprimir o menu (onde cada item tem um número para o cliente fazer o pedido, um nome que descreve o produto e um preço - note que por enquanto não é necessário encapsular estes valores, somente imprimi-los) e um método estático `fazPedido` para pedir ao usuário para que entre um valor correspondente ao número de um dos itens mostrados no menu. O método `fazPedido` deverá retornar o valor do item escolhido.

**Exercício 6.20:** ⋆⋆

Escreva uma classe que encapsule uma carta de baralho, com um valor que represente o valor da carta, de um (ás) a treze (rei), e outro valor correspondente ao naipe (1 = ouros, 2 = paus, 3 = copas e 4 = espadas). Escreva nesta classe um método que imprima o nome da carta por extenso, usando a instrução `switch`.

**Exercício 6.21:** ⋆⋆⋆

Escreva um programa em Java que simule uma calculadora bem simples. Este programa deve ler dois valores de ponto flutuante do teclado e um caracter, correspondente a uma das operações básicas (`+`, `-`, `*` ou `/`), calcular a operação e imprimir o resultado. O programa deve considerar divisões por zero como sendo erros, e imprimir uma mensagem adequada.

**Exercício 6.22:** ⋆⋆⋆

Modifique o programa do exercício 6.21 para que caso o segundo valor entrado seja igual a zero, o programa não permita a escolha da operação divisão.

**Exercício 6.23:** ⋆⋆⋆⋆

Modifique a classe `Retangulo` (exercício 2.50) para que esta contenha um método adicional `calculaIntersecção`, que recebe como argumento uma outra instância da própria classe `Retangulo` e calcule um retângulo que é a intersecção do retângulo encapsulado com o passado como argumento, retornando uma nova instância da classe `Retangulo` correspondente à intersecção. *Dicas:* os pontos do retângulo-intersecção podem ser calculados com regras simples, implementadas através de `if`s encadeados. Nem sempre existe intersecção entre dois retângulos: considere a figura 6.1, que no lado esquerdo existem dois retângulos (mostrados em cores diferentes) que tem intersecção, e no lado direito, dois que não tem. No caso de não existir intersecção, o método deve retornar `null`. Veja também o exercício 2.65.

**Exercício 6.24:** ⋆⋆⋆⋆

Usando o exercício 6.23 como base, escreva mais dois métodos na classe `Retangulo` que recebam uma instância da classe `Retangulo` como argumento e retornem, separadamente, a área e o perímetro da intersecção do retângulo encapsulado e do retângulo passado como argumento.

**Exercício 6.25:** ⋆⋆⋆⋆

Usando o exercício 6.23 como base, escreva métodos sobrecarregados que calculem os retângulos correspondentes à intersecção do retângulo encapsulado na classe `Retangulo` com duas, três e quatro instâncias da própria classe `Retangulo`, passadas como argumentos para o método `calculaIntersecção`.

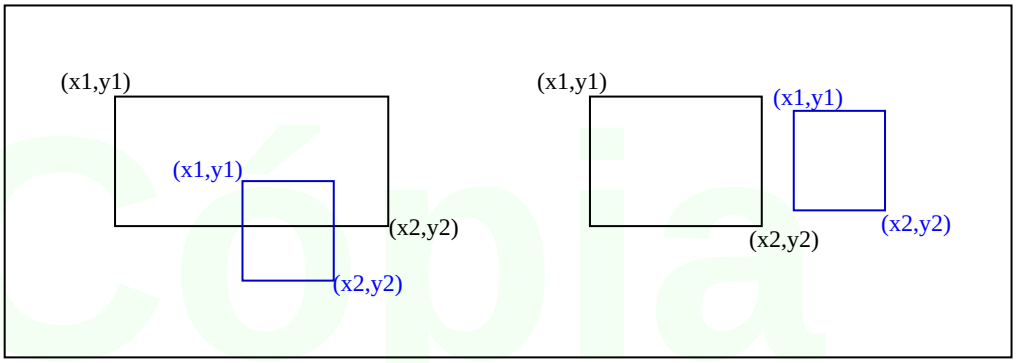



Figura 6.1: Intersecção de dois retângulos.

## 6.5 Exercícios complementares do capítulo 6

**Exercício 6.26:** ⋆⋆

Modifique a classe `ContaBancariaSimplificada` (exercício 2.53) para que o método `retira` permita a retirada de valores de contas especiais de forma que o saldo negativo máximo seja 1000 reais, ou seja, não permitindo que contas especiais fiquem com débito maior do que 1000 reais,

**Exercício 6.27:** ⋆⋆

O método `calculaMensalidade` da classe `RegistroAcademicoSemConstrutor` tem um ponto ineficiente: mesmo que o valor do campo `percentualDeCobrança` seja igual a zero, vários comandos `if` serão executados para verificar qual deve ser a mensalidade dependendo do código do curso. Modifique o fluxo do método para que este seja mais eficiente.

**Exercício 6.28:** ⋆⋆⋆

Reescreva a classe `Comparavel` (listagem 6.1) para que todos os seus métodos sejam estáticos, e o valor sendo comparado seja passado como argumento adicional para os métodos. Que vantagens e desvantagens você vê nas duas abordagens ?

**Exercício 6.29:** ⋆⋆⋆

Escreva uma versão da classe `RegistroAcademico` (usando a classe `RegistroAcademicoSemConstrutor` na listagem 4.1) para que esta tenha um construtor que receba o nome do aluno, um número de matrícula, um código do curso e um percentual de cobrança. O construtor deverá garantir que o percentual de cobrança esteja entre zero e cem por cento, assumindo que se um valor inválido for passado, o valor será considerado como sendo igual a cem.
O construtor também deverá verificar se o código do curso é um dos valores reconhecidos (que podem ser 34 para `"Ciência da Computação"`, 39 para `"Engenharia da Computação"`, 41 para `"Arquitetura"`, 43 para `"Engenharia Civil"` e 45 para `"Engenharia Elétrica"`). Qualquer valor diferente destes deverá ser considerado zero.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 6.30:** ⋆⋆⋆

Escreva para a classe `RegistroAcademico` (exercício 6.29) um método `toString` que retorne o nome, número de matrícula e nome do curso do aluno encapsulado pela classe. Para retornar o nome do aluno considere a lista de cursos mostrada no exercício 6.29. Se o código do curso for zero (indicando inválido), considere o nome do curso como sendo "`Incorreto`".

# Capítulo 7

# Estruturas de Decisão e Controle - Repetição

## 7.1 Estruturas de repetição ou iteração

*iteração*

As instruções `if/else` e `switch`, vistas no capítulo anterior, permitem que o fluxo de execução de um método ou programa seja alterado dependendo de uma condição, mas sempre de cima para baixo (no sentido do fluxo de execução), ou seja, executando trechos de código e deixando de executar outros. Não é possível, com estas instruções, repetir parte do código que foi executado anteriormente ou *iterar*.

Vemos a necessidade de repetir, iterar ou contar em muitos modelos e aplicações: o fatorial de um número é calculado multiplicando-se os valores *de* um *até* o valor especificado; uma seqüência de caracteres em uma String representando parte do DNA é comparada com outra se tomando os caracteres *um por um*, *do* primeiro *até* o último; uma impressão de relatórios sobre bolsistas pode ser feita abrindo-se um arquivo, lendo os dados dos bolsistas *um por um*, *enquanto* existirem informações no arquivo; uma autenticação de uso de programas por meio de senha pode ser feita pedindo-se ao usuário que entre a senha *até que* a senha correta seja entrada *ou* o número de tentativas esgotado.

A capacidade de repetição de trechos de programas ou de métodos é uma das características de linguagens de programação que tornam os computadores mais úteis: dada uma tarefa ou rotina, esta pode ser repetida inúmeras vezes de forma automática. A repetição por si própria não é muito útil, a não ser que diferentes dados sejam processados a cada iteração da repetição. Considere a tarefa de somar muitos valores ou procurar palavras em um texto longo ou pesquisar vários registros de um banco de dados - tarefas tediosas se feitas por um ser humano, mas que podem ser facilmente implementadas e repetidas por um computador.

*laços*

As estruturas de repetição em Java são chamadas coletivamente de *laços*. Em suas formas básicas elas consideram uma condição que determina se o laço deve ser executado ou não, e executam um ou mais comandos enquanto a condição especificada for válida ou verdadeira.

*variável de controle*

Uma tarefa inerente à execução de laços em Java é a modificação de variáveis que controlam a execução dos laços (chamadas *variáveis de controle*). Estas variáveis podem refletir o estado da execução de um laço, por exemplo, representando um contador que contará quantas vezes a repetição será feita, ou verificando se uma condição já foi cumprida para que o laço termine. Um caso especial de variáveis de controle são contadores, vistos na subseção seguinte.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

### 7.1.1 Contadores

*contadores*

*Contadores* são variáveis que recebem um valor inicial e são modificadas a cada iteração de uma estrutura de repetição em Java. Apesar de serem basicamente variáveis comuns, contadores merecem uma atenção à parte pelos mecanismos de modificação de seus valores e cuidados especiais a serem tomados.

Contadores podem ser variáveis de qualquer tipo numérico. Estas variáveis devem ser inicializadas com um valor inicial, alteradas a cada iteração do laço e verificadas para ver se um valor final foi alcançado. Por exemplo, para fazer uma estrutura que conte de um até dez, usaremos um contador que receberá o valor inicial de um, sendo acrescido de um em um até que o valor deste contador seja igual a dez, interrompendo então a repetição do laço.

Valores dos contadores são alterados através da atribuição do resultado de uma operação à variável que representa o contador. Geralmente estas operações envolvem a própria variável contadora, como por exemplo, `linha = linha + 1;`, que fará com que a variável `linha` receba o valor que ela continha acrescido de um. Java tem operadores especiais para a modificação de variáveis usando a própria variável, que são:

*operador ++*

- `++` Este operador, quando aplicado a uma variável dos tipos inteiros ou ponto flutuante, incrementará o valor da variável em um. Por exemplo, ao final do código `double a = 12.5; a++;` , a variável `a` valerá 13.5. O comando `a++` poderia ser escrito como `a = a + 1`.
  O operador `++` pode ser usado antes ou depois do nome da variável que incrementará. Aparentemente não existe diferença entre aplicar o operador antes ou depois da variável, mas quando lembramos que o operador, além de modificar o conteúdo da variável, retorna o valor modificado, a diferença fica patente. Se a variável `cont` valer `17`, ao final do código `val = cont++;` a variável `val` valerá `17`, pois o operador retornou o valor da variável `cont` antes de incrementá-lo. Por outro lado, se a variável `cont` valer `17`, ao final do código `val = ++cont;` ambas as variáveis valerão `18`.

*operador +=*

- `+=` Este operador, quando aplicado a uma variável, incrementará o valor da variável usando o argumento para o incremento. Por exemplo, ao final do código `short val = 800; val += 200;` , a variável `val` valerá 1000. O comando `val += 200` poderia ser escrito como `val = val + 200`.

*operador --*

- `--` Este operador, quando aplicado a uma variável, decrementará o valor da variável em um. Por exemplo, ao final do código `int z = 1982; z--;` , a variável `z` valerá 1981. O comando `z--` poderia ser escrito como `z = z - 1`. O operador `--`, como o operador `++`, pode ser aplicado antes ou depois da variável que será modificada, com as mesmas regras mostradas para o operador `++`.

*operador -=*

- `-=` Este operador, quando aplicado a uma variável, decrementará o valor da variável usando o argumento para o incremento. Por exemplo, ao final do código `float med = 49.5f; med -= 11.04;` , a variável `med` valerá 38.46. O comando `med -= 11.04` poderia ser escrito como `med = med - 11.04`.

*operador *=*

- `*=` Este operador, quando aplicado a uma variável, multiplicará o valor presente da variável pelo argumento passado, colocando o resultado na variável. Por exemplo, ao final do código `double j = 732.5; j *= 4;` , a variável `j` valerá 2930. O comando `j *= 4` poderia ser escrito como `j = j * 4`.

*operador /=*

- `/=` Este operador, quando aplicado a uma variável, dividirá o valor presente da variável pelo argumento passado, colocando o resultado na variável. Por exemplo, ao final do código `int o = 2000; o /= 12;` , a variável `o` valerá 166 (note que a divisão foi feita com números inteiros). O comando `o /= 12` poderia ser escrito como `o = o / 12`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

A alteração de contadores pode levar a problemas quando os valores modificados são muito grandes ou pequenos para ser representados pelos seus tipos. Por exemplo, lembrando que um valor do tipo `short` pode representar valores entre -32768 e 32767, ao final do código `short cont = 32767; cont += 1;` a variável `cont` valerá `-32768` - se uma variável de certo tipo numérico inteiro é acrescido de forma que o resultado é maior do que pode ser armazenado, a variável recebe o menor valor representável mais a diferença menos um. Pode-se considerar que para valores do tipo `short`, o compilador Java considera que eles estão na seqüência crescente `...32766, 32767, -32768, -32767...`. Similarmente, valores do tipo `byte` são considerados como estando na seqüência decrescente `...-126, -127, -128, 127, 126, 125...`.

Este problema é conhecido como *overflow* - infelizmente o compilador Java não informa este tipo de erro em potencial. Cabe ao programador tomar cuidado para usar o tipo de variável mais adequado para seus contadores (veja a seção 2.3.1 e a tabela 2.2 para informações sobre os limites dos valores armazenáveis nos tipos numéricos). *Overflow* também acontece com valores de ponto flutuante, exceto que um tratamento mais correto é dado: ao final do código `float valor = 3.4e38f; valor *= 2;` a variável `valor` valerá `Infinity`, um valor especial que existe para os tipos `double` e `float`.

overflow

De forma similar ao *overflow*, erros potenciais de *underflow* podem ocorrer quando o tipo de ponto flutuante não é capaz de representar um número muito pequeno (próximo de zero) - o compilador arredondará estes valores para zero. De novo, o compilador não indicará erros, então cabe ao programador verificar estas condições em potencial.

underflow

### 7.1.2 O laço `while`

Uma das estruturas que permite repetição em programas e métodos em Java é a instrução `while`, que repete um comando ou bloco de comandos enquanto uma condição for verdadeira. A forma básica desta instrução é `while(condição) comando_ou_bloco;`. A condição deve ser um valor booleano ou expressão cujo resultado seja booleano, e o comando ou bloco associado ao laço pode ser um único comando, que deve ser terminado com um ponto-e-vírgula, ou um bloco de comandos, que deve ser cercado por chaves (`{` e `}`). O bloco ou comando associado ao laço será repetido enquanto o valor booleano avaliado pela instrução `while` a cada iteração for verdadeiro.

while

Se o argumento para a instrução `while` for inicialmente `false`, o comando ou bloco de comandos associado não será executado nem mesmo uma vez. Para que o laço eventualmente termine, o valor ou expressão avaliado pelo `while` deve eventualmente receber o valor `true`. Alternativamente, o bloco associado à instrução `while` pode ser terminado abruptamente por um comando `break` (geralmente executado por um `if`). A classe `DemoWhile` na listagem 7.1 demonstra vários usos do laço `while`.

break *em laço* while

Listagem 7.1: A classe `DemoWhile`, que demonstra vários usos do laço `while`.

```
1  /**
2   * A classe DemoWhile, que demonstra vários exemplos de uso de laços while.
3   * ESTA CLASSE NÃO PODE SER EXECUTADA NORMALMENTE POR CAUSA DE ERROS INTENCIONAIS.
4   */
5  class DemoWhile
6    {
7    /**
8     * O método main permite a execução desta classe. Este método contém quatro exemplos
9     * de laços while, sendo que um deles é um laço "infinito". Se a classe for
10    * executada, o usuário deverá interromper o programa quando o laço "infinito" for
```

```
11    * executado.
12    * @param argumentos os argumentos que podem ser passados para o método via linha
13    *        de comando, mas que neste caso serão ignorados.
14    */
15   public static void main(String[] argumentos)
16     {
17     // Primeiro exemplo de laço while: um contador simples
18     double valor = 1; // um valor inicial que será modificado pelo laço
19     while(valor < 2000) // enquanto este valor for menor que 2000
20       {
21       System.out.println(valor); // imprimimos o valor
22       valor *= 2; // o valor é dobrado e a iteração continua enquanto for < 2000
23       }
24     // Segundo exemplo de laço while: usamos o break para sair do laço
25     char letra = 'A'; // vamos imprimir todos os caracteres maiúsculos
26     while(true) // aparentemente este laço será executado infinitamente
27       {
28       System.out.print(letra); // imprimimos a letra sem quebrar a linha
29       letra++; // o valor da letra é incrementado
30       if (letra > 'Z') // chegamos ao fim dos caracteres maiúsculos
31         {
32         System.out.println(); // imprimimos a quebra de linha
33         break; // e saímos do laço
34         }
35       }
36     // Terceiro exemplo de laço while: um laço que não chega a ser executado
37     int contador = 100; // vamos contar de 100 a 50, decrementando o contador
38     while(contador < 50) // mas o comparador foi criado incorretamente e será igual
39       {                  // a false antes da primeira iteração !
40       contador--; // decrementamos o contador
41       System.out.print(contador+" ");
42       }
43     // Quarto exemplo de laço while: um laço "infinito" por distração do programador.
44     // Vamos calcular os fatores de 2 que sejam menores que 4611686018427387904 (2^62).
45     long fatorDe2 = 1; // valor inicial
46     while(fatorDe2 < 4611686018427387904L)
47       {
48       System.out.println(fatorDe2); // o valor é impresso mas não é modificado:
49       }                             // o laço executará "infinitamente".
50
51     } // fim do método main
52
53   } // fim da classe DemoWhile
```

O laço `while` nas linhas 47 a 50 na listagem 7.1 é dito “infinito” porque não existem condições para o seu término natural, no entanto, é possível para o operador, o sistema operacional ou a máquina virtual interromper o programa quando este estiver sendo executado.

Um exemplo mais prático de laço `while` é dado pela classe `EscolhaComWhile`, que contém um método que repete um pedido de entrada de dados via teclado até que o valor entrado esteja entre uma faixa de valores conhecidos. A classe `EscolhaComWhile` é mostrada na listagem 7.2.

Listagem 7.2: A classe `EscolhaComWhile`, que encapsula um mecanismo de escolha de valores (usando um bloco `while`).

```
1  /**
2   * A classe EscolhaComWhile, que encapsula o mecanismo de escolha de um número que
3   * deve estar entre dois números também encapsulados na classe. O método que pede ao
4   * usuário para fazer a escolha a entrada do valor escolhido (usando um bloco while)
5   * até que o valor esteja na faixa esperada.
6   */
7  class EscolhaComWhile // declaração da classe
8    {
9   /**
10   * Declaração dos campos da classe
```

```
11   */
12   private short início,fim; // a faixa de valores válidos para escolha
13
14  /**
15   * O construtor para a classe EscolhaComWhile, que receberá como argumentos os dois
16   * valores extremos (inicial e final ou menor e maior), que determinarão a faixa de
17   * valores dos quais poderemos escolher.
18   * @param i o valor inicial (ou menor valor da faixa)
19   * @param f o valor final (ou maior valor da faixa)
20   */
21   EscolhaComWhile(short i,short f)
22     {
23     início = i;
24     fim = f;
25     }
26
27  /**
28   * O método escolhe pergunta um valor ao usuário até que o valor entrado esteja
29   * entre os valores inicial e final (inclusive).
30   * @return o valor entrado pelo usuário
31   */
32   public short escolhe()
33     {
34     short valorEntrado = 0; // devemos inicializar esta variável com qualquer valor
35     boolean valorSatisfazCondições = false; // inicialmente consideraremos que o
36                                             // valor entrado não satisfaz as condições
37                                             // (não está dentro da faixa dos valores
38                                             // inicial e final)
39     while(!valorSatisfazCondições) // enquanto o valor não satisfizer as condições
40       {
41       System.out.print("Entre um valor entre "+início+" e "+fim+":");
42       valorEntrado = Keyboard.readShort();
43       if ((valorEntrado >= início) &&
44           (valorEntrado <= fim))        // se o valor entrado estiver na faixa, a
45         valorSatisfazCondições = true;  // variável de controle receberá true
46       }
47     return valorEntrado; // retornamos o valor entrado
48     } // fim do método escolhe
49
50   } // fim da classe EscolhaComWhile
```

O método `escolhe` na classe `EscolhaComWhile` repete o trecho de código que pede ao usuário que entre um valor até que uma condição seja satisfeita; no caso, que a variável de controle booleana `valorSatisfazCondições` valha `true`. Dentro do bloco associado à instrução `while` temos condições que podem alterar o valor da variável de controle de seu valor original `false` para `true`.

No exemplo dado, a instrução `while` e seu bloco de comandos associado poderiam ser escritos de forma diferente, por exemplo, verificando a condição de continuação dentro do `while` sem precisar usar a variável de controle, mas a forma escolhida faz o método ficar mais claro.

### 7.1.3 O laço `do-while`

O bloco `while` do método `escolhe` na classe `EscolhaComWhile` (listagem 7.2) usa um artifício para que o laço seja executado ao menos uma vez: fazendo com que o valor da variável de controle `valorSatisfazCondições` seja inicialmente igual a `false`, garantimos que o laço será executado ao menos uma vez. Idealmente seria melhor executar uma vez a leitura do dado, e só continuar se o dado estiver fora da faixa de valores válidos.

Java ofereçe outro tipo de laço que é executado enquanto uma condição for verdadeira, mas `do-while`

garante que o bloco associado ao laço será executado ao menos uma vez: o laço `do-while`, cuja forma é `do { comando_ou_bloco } while(condição);`. A condição deve ser um valor booleano ou expressão cujo resultado seja booleano, e a expressão avaliada pela instrução `while` deve ser seguida de um ponto-e-vírgula. Diferentemente do laço `while`, a sintaxe do laço `do-while` exige que o bloco de comandos associados esteja entre as chaves (`{` e `}`).

A classe `EscolhaComDoWhile`, mostrada na listagem 7.3, mostra como o laço `while` do método `escolhe` na listagem 7.2 pode ser reescrito para usar um laço `do-while`.

Listagem 7.3: A classe `EscolhaComDoWhile`, que encapsula um mecanismo de escolha de valores (usando um bloco `do-while`).

```
1  /**
2   * A classe EscolhaComDoWhile, que encapsula o mecanismo de escolha de um número
3   * que deve estar entre dois números também encapsulados na classe. O método que
4   * pede ao usuário para fazer a escolha repete a entrada do valor escolhido
5   * (usando um bloco do-while) até que o valor esteja na faixa esperada.
6   */
7  class EscolhaComDoWhile // declaração da classe
8    {
9   /**
10   * Declaração dos campos da classe
11   */
12   private short início,fim; // a faixa de valores válidos para escolha
13
14  /**
15   * O construtor para a classe EscolhaComDoWhile, que receberá como argumentos os dois
16   * valores extremos (inicial e final ou menor e maior), que determinarão a faixa de
17   * valores dos quais poderemos escolher.
18   * @param i o valor inicial (ou menor valor da faixa)
19   * @param f o valor final (ou maior valor da faixa)
20   */
21   EscolhaComDoWhile(short i,short f)
22     {
23     início = i;
24     fim = f;
25     }
26
27  /**
28   * O método escolhe pergunta um valor ao usuário até que o valor entrado esteja
29   * entre os valores inicial e final (inclusive).
30   * @return o valor entrado pelo usuário
31   */
32   public short escolhe()
33     {
34     short valorEntrado;
35     boolean valorSatisfazCondições; // inicialmente pode ter valor indefinido
36     do // execute o seguinte bloco
37       {
38       System.out.print("Entre um valor entre "+início+" e "+fim+":");
39       valorEntrado = Keyboard.readShort();
40       if ((valorEntrado >= início) &&
41           (valorEntrado <= fim))        // se o valor entrado estiver na faixa, a
42         valorSatisfazCondições = true;  // variável de controle receberá true
43       else valorSatisfazCondições = false; // senão, receberá false - neste caso,
44       }                                    // devemos atribuir um valor à variável
45     while(!valorSatisfazCondições); // execute enquanto esta condição for verdadeira
46     return valorEntrado; // retornamos o valor entrado
47     } // fim do método escolhe
48
49   } // fim da classe EscolhaComDoWhile
```

Um ponto interessante da classe `EscolhaComDoWhile` (listagem 7.3) é que não é necessário dar um valor inicial para a variável de controle `valorSatisfazCondições`: a variável somente será

avaliada no final do laço, então só é necessário garantir que ela receberá algum valor dentro do laço.

A classe `JogoDeAdivinhacao`, na listagem 7.4, mostra outro método que usa o laço `do-while`. Esta classe encapsula o funcionamento de um jogo simples de adivinhação, onde um jogador deverá adivinhar um número secreto, auxiliado por dicas dadas pela classe depois de cada tentativa.

Listagem 7.4: A classe `JogoDeAdivinhacao`, que implementa um jogo simples de adivinhação de números.

```
/**
 * A classe JogoDeAdivinhacao, que implementa um jogo simples de adivinhação de
 * números. Um número será escolhido e o usuário deverá adivinhar este número com
 * dicas da classe.
 * No momento esta classe é bem simples: o valor a ser adivinhado deve ser fornecido
 * pelo programa que criar instâncias da classe. Futuramente veremos como usar valores
 * aleatórios para isto.
 */
class JogoDeAdivinhacao // declaração da classe
  {
 /**
  * Declaração dos campos da classe
  */
  private int valor; // o valor a ser adivinhado

 /**
  * O construtor da classe simplesmente inicializa o campo privado com o valor
  * passado como argumento.
  * @param val o valor passado para inicialização do campo valor
  */
  JogoDeAdivinhacao(int val)
    {
    valor = val;
    } // fim do construtor

 /**
  * O método tenta pergunta ao usuário um valor e mostra se o valor encapsulado é
  * maior ou menor do que o valor passado. A pergunta é feita até que vinte tentativas
  * sejam feitas ou até que o usuário tenha acertado o valor.
  */
  public void tenta()
    {
    byte tentativas = 20; // o número máximo de tentativas
    byte contador = 1; // o número de tentativas até agora
    System.out.println("Tente adivinhar o número secreto em "+tentativas+
                       " tentativas.");
    do // execute o bloco seguinte...
      {
      System.out.print("Tentativa número "+contador+":");
      int estaTentativa = Keyboard.readInt();
      if (estaTentativa == valor) // acertou o valor !
        break; // simplesmente sai do laço
      else if (estaTentativa < valor)
        System.out.println("O número "+estaTentativa+" é menor que o número secreto !");
      else
        System.out.println("O número "+estaTentativa+" é maior que o número secreto !");
      contador++;
      }
    while(contador <= tentativas); // enquanto o número de tentativas não for alcançado
    // Aqui sabemos que ou o usuário acertou o número ou as tentativas se esgotaram.
    // Basta verificar o valor do contador: se for maior que o número de tentativas,
    // então o usuário não conseguiu adivinhar o número.
    if (contador > tentativas)
      System.out.println("Você não acertou !");
    else
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
56        System.out.println("Parabéns, você acertou o número ("+valor+") !");
57      }  // fim do método tenta
58
59    } // fim da classe JogoDeAdivinhacao
```

O método `tenta` na classe `JogoDeAdivinhacao` controla a iteração do jogo. Existem duas condições para o término do jogo: ou o número de tentativas é esgotado sem que o jogador acerte o número secreto ou o jogador acerta o número. O controle do número de tentativas é feito pelo laço `do-while`, enquanto que um `if` acompanhado de `break` terminará o laço caso o número correto seja entrado. Após o término do laço por uma das duas condições, o número de tentativas é avaliado por um `if` para verificar se o laço terminou por número de tentativas esgotado ou acerto do número secreto.

A classe `DemoJogoDeAdivinhacao`, mostrada na listagem 7.5, cria e usa uma instância da classe `JogoDeAdivinhacao`.

Listagem 7.5: A classe `DemoJogoDeAdivinhacao`, que demonstra o uso de uma instância da classe `JogoDeAdivinhacao`.

```
1  /**
2   * A classe DemoJogoDeAdivinhacao, que demonstra o uso de uma instância da classe
3   * JogoDeAdivinhacao.
4   */
5  class DemoJogoDeAdivinhacao // declaração da classe
6    {
7    /**
8     * O método main permite a execução desta classe. Este método cria uma instância da
9     * classe JogoDeAdivinhacao e executa o método tenta para permitir a interação com o
10    * usuário.
11    * @param argumentos os argumentos que podem ser passados para o método via linha
12    *        de comando, mas que neste caso serão ignorados.
13    */
14   public static void main(String[] argumentos)
15     {
16     JogoDeAdivinhacao novoJogo = new JogoDeAdivinhacao(237865);
17     novoJogo.tenta();
18     } // fim do método main
19
20   } // fim da classe DemoJogoDeAdivinhacao
```

### 7.1.4 O laço `for`

Tanto a estrutura `while` quando a `do-while` podem ser usadas para implementar repetição controlada por contadores. Para isto, devemos criar uma variável de controle que deve ser inicializada antes do laço, ter seu valor modificado durante o laço e ter seu valor comparado com outro para decidir se o bloco associado ao laço deve ser repetido ou não.

`for` Java tem uma estrutura especializada para a implementação de repetição controlada por contadores, que agrupa a inicialização, modificação e comparação da variável de controle em uma única instrução, ao invés de espalhadas pelo código. Esta instrução chama-se `for`, cuja forma básica é `for(inicialização;verificação_de_condições;atualização) comando_ou_bloco`, onde `inicialização` pode conter os comandos que devem ser executados antes do início do laço, `verificação_de_condições` geralmente é uma expressão booleana que é verificada antes de cada iteração e `atualização` pode conter comandos que serão executados após o final de cada iteração. O comando ou bloco associado ao laço pode ser um único comando, que deve ser terminado com um ponto-e-vírgula, ou um bloco de comandos, que deve ser cercado por chaves (`{` e `}`).

O laço `for`, quando usado para contadores simples, usa a expressão `inicialização` para inicializar a variável de controle, a expressão `verificação_de_condições` para verificar se a variável de controle já atingiu o valor esperado e a expressão `atualização` para modificar o valor da variável de controle ao final de cada iteração. A classe `DemoFor` na listagem 7.6 demonstra vários usos do laço `for` como contador.

Listagem 7.6: A classe `DemoFor`, que demonstra vários usos do laço `for`.

```
/**
 * A classe DemoFor, que demonstra vários exemplos de uso de laços for como
 * contadores.
 * ESTA CLASSE NÃO PODE SER EXECUTADA NORMALMENTE POR CAUSA DE ERROS INTENCIONAIS.
 */
class DemoFor
  {
  /**
   * O método main permite a execução desta classe. Este método contém cinco exemplos
   * de laços for, sendo que um deles é um laço "infinito". Se a classe for executada,
   * o usuário deverá interromper o programa quando o laço "infinito" for executado.
   * @param argumentos os argumentos que podem ser passados para o método via linha
   *        de comando, mas que neste caso serão ignorados.
   */
  public static void main(String[] argumentos)
    {
    // Primeiro exemplo de laço for: um contador simples que conta de 0 a 99
    int contador;
    for(contador = 0;contador < 100; contador++) // nunca alcança 100
      {
      System.out.println(contador);
      }
    // Segundo exemplo de laço for: um contador que conta do maior para o menor valor,
    // com decrementos. A variável contador, declarada anteriormente, é reutilizada.
    for(contador = 200;contador >= 0; contador -= 10) // de 200 a 0 de -10 em -10
      {
      System.out.println(contador);
      }
    // Terceiro exemplo de laço for: um contador que não chega a ser executado pois
    // a expressão de verificação do valor final já é avaliada como false no início do
    // laço.
    for(double controle = 0;controle < 0; controle += 3.5)
      {
      System.out.println(controle);
      }
    // Quarto exemplo de laço for: um laço aparentemente "infinito" que eventualmente
    // será terminado por um comando break associado à uma instrução if.
    double início     = 100;
    double fim        = 200;
    double incremento = 2.5;
    for(;;) // equivalente a dizer while(true)
      {
      System.out.println(início);
      if (início >= fim) break; // termina se início for maior que fim
      início += incremento; // se não terminou, incrementa
      }
    // Quinto exemplo de laço for: um laço "infinito" por distração do usuário
    for(contador = 0;contador < 1000; contador+=2) // de 0 a 1000, somente os pares
      {
      System.out.println(contador);
      contador = 0; // a variável de controle é reinicializada !
      }
    } // fim do método main

  } // fim da classe DemoFor
```

Quando variáveis de controle são usadas somente para controlar o contador de um laço `for`, é



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

comum declarar as variáveis dentro da expressão `inicialização`, como mostrado na linha 32 da listagem 7.6. O escopo da variável fica sendo o laço, a variável não será mais definida e não poderá ser usada assim que o laço for encerrado.

A classe `ProbabilidadeBasica`, mostrada na listagem 7.7, contém os métodos `combinações` e `permutações`, que calculam respectivamente combinações e permutações, usando as fórmulas $comb = Q!/(T!(Q-T)!)$ para calcular o número *comb* de combinações que se pode ter tendo $Q$ valores e tomando-se $T$ valores de cada vez e $perm = Q!/(Q-T)!$ para calcular o número de permutações *perm* que se pode ter tendo $Q$ valores e tomando-se $T$ valores de cada vez. Ambos os métodos usam o cálculo do fatorial de um número, definido por $X! = 1*2*3*...*X$ que é implementado no método `fatorial`, que usa um laço `for` para o cálculo.

Listagem 7.7: A classe `ProbabilidadeBasica`, que contém alguns métodos estáticos para cálculos básicos de probabilidades.

```
1  /**
2   * A classe ProbabilidadeBasica, que contém alguns métodos estáticos para cálculos
3   * básicos de probabilidades.
4   */
5  class ProbabilidadeBasica // declaração da classe
6    {
7   /**
8    * O método fatorial calcula o fatorial de um valor inteiro, retornando o resultado
9    * como um valor do tipo double (para evitar overflow)
10   * @param val o valor para o qual calcularemos o fatorial
11   * @return o fatorial do valor passado como argumento
12   */
13   public static double fatorial(int val)
14     {
15     double fatorial = 1; // o acumulador para o fatorial
16     for(int i=1;i<=val;i++) // contamos de um até o valor
17       fatorial = fatorial*i; // e multiplicamos cumulativamente
18     return fatorial; // retornamos o resultado do fatorial
19     }
20
21  /**
22   * O método combinações calcula o número de combinações que podem ser obtidas
23   * quando T elementos são tomados de um grupo de Q elementos.
24   * @param Q o número total de elementos
25   * @param T o tamanho do subgrupo de Q.
26   * @return o número de combinações de Q elementos tomados T a T
27   */
28   public static double combinações(int Q,int T)
29     {
30     return fatorial(Q)/(fatorial(T)*fatorial(Q-T));
31     }
32
33  /**
34   * O método permutações calcula o número de permutações que podem ser obtidas
35   * quando T elementos são tomados de um grupo de Q elementos.
36   * @param Q o número total de elementos
37   * @param T o tamanho do subgrupo de Q.
38   * @return o número de permutações de T elementos tomados de um grupo de Q elementos.
39   */
40   public static double permutações(int Q,int T)
41     {
42     return fatorial(Q)/fatorial(Q-T);
43     }
44
45   } // fim da classe ProbabilidadeBasica
```

A classe `DemoProbabilidadeBasica` (listagem 7.8) demonstra usos dos métodos da classe `ProbabilidadeBasica`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem 7.8: A classe `DemoProbabilidadeBasica`, que demonstra o uso dos métodos estáticos da classe `ProbabilidadeBasica`.

```
/**
 * A classe DemoProbabilidadeBasica, que demonstra o uso dos métodos estáticos da
 * classe ProbabilidadeBasica.
 */
class DemoProbabilidadeBasica // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método chama os métodos
  * estáticos da classe ProbabilidadeBasica para sua demonstração.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Problema de combinações: quantos conjuntos diferentes de três pessoas
    // podemos criar de um grupo de seis pessoas ?
    System.out.println(ProbabilidadeBasica.combinações(6,3));   // 20
    // Problema de combinações: quantos times diferentes de futebol (titulares) podemos
    // fazer se temos vinte e dois jogadores ?
    System.out.println(ProbabilidadeBasica.combinações(22,11)); // 705432
    // Problema de permutações: de quantas maneiras diferentes podemos sentar seis
    // pessoas em uma mesa ?
    System.out.println(ProbabilidadeBasica.permutações(6,6));   // 720
    // Problema de permutações: e se tivermos dez pessoas mas só cinco assentos ?
    System.out.println(ProbabilidadeBasica.permutações(10,5));  // 30240
    } // fim do método main

  } // fim da classe DemoProbabilidadeBasica
```

## 7.2 Introdução à Recursão

*recursão*
*recursividade*

Recursão (ou recursividade) é uma técnica de programação onde um algoritmo, para ser executado, chama o próprio algoritmo para resolver uma parte menor do problema, desta forma quebrando o problema em pedaços menores (esperançosamente com soluções mais fáceis ou rápidas).

Um exemplo clássico de recursão pode ser dado por um algoritmo que calcule a somatória dos $N$ primeiros números inteiros. A somatória dos $N$ primeiros números inteiros é igual a $N$ mais a somatória dos $N-1$ primeiros números inteiros, que por sua vez, é igual a $N-1$ mais a somatória dos $N-2$ primeiros números inteiros, e assim sucessivamente, de forma que a somatória a ser calculada é sempre mais simples do que se a recursão não fosse usada. Evidentemente uma condição de parada nesta seqüência se faz necessária: quando $N$ for zero, a somatória será zero sem que seja necessário calcular a soma de de $N$ com a somatória dos $N-1$ números anteriores. Esta condição de parada, usada quando não vale à pena quebrar o problema a ser resolvido em pedaços ainda menores, é essencial para que o algoritmo recursivo não seja executado eternamente.

*condição de parada*

A implementação de recursão usando linguagens de programação sugere o uso de métodos que serão chamados com argumentos que delimitam ou simplificam cada vez mais o algoritmo a ser resolvido, e que quando o algoritmo não puder ser mais simplificado, retornem uma operação ou valor qualquer. No exemplo da somatória, o algoritmo poderia ser descrito como:

```
somatória(N)
  se N == 0, retorne 0
  senão, retorne N + somatória(N-1)
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Na descrição acima vemos que a subrotina ou método `somatória` é chamada repetidamente a partir de si mesma, cada vez com argumentos que farão a tarefa mais simples, a não ser que o valor de *N* seja zero. É importante notar que sem a condição de parada (`se N == 0`) o algoritmo seria repetido infinitamente, o que certamente não é o desejado.

A classe `SomatoriaRecursiva` contém um método `main` que mostra o cálculo da somatória dos primeiros *N* números inteiros, onde *N* é entrado pelo teclado.

Listagem 7.9: A classe `SomatoriaRecursiva`, que demonstra o cálculo da somatória recursiva dos primeiros *N* números inteiros.

```
/**
 * A classe SomatoriaRecursiva, que demonstra o cálculo da somatória dos primeiros
 * N números inteiros através da chamada recursiva de um método estático.
 */
class SomatoriaRecursiva
  {
  /**
   * O método main permite a execução desta classe. Este método calcula a somatória
   * dos N primeiros números inteiros, onde N é entrado pelo teclado. O cálculo é
   * feito de forma recursiva.
   * @param argumentos os argumentos que podem ser passados para o método via linha
   *        de comando, mas que neste caso serão ignorados.
   */
  public static void main(String[] argumentos)
    {
    System.out.println("Este programa calcula a somatória dos N primeiros "+
                       "números usando um método recursivo.");
    System.out.print("Entre o valor de N:");
    int n = Keyboard.readInt();
    long resultado = somatória(n);
    System.out.println("A somatória é "+resultado+".");
    } // fim do método main

 /**
  * O método somatória calcula a soma dos N primeiros números inteiros recursivamente.
  * O valor de N é passado como argumento, e a cada passo o método calcula a
  * somatória dos N-1 primeiros números, somando este resultado ao valor de N, a não
  * ser que N seja igual a zero.
  * Este método é menos complicado do que aparenta: muitos comandos de impressão
  * foram adicionados para maior clareza na execução do programa, e podem ser
  * retirados sem problemas.
  * @param N o número de inteiros que devem ser somados
  * @return a somatória dos N primeiros valores
  */
  static long somatória(int N)
    {
    System.out.println("Vamos calcular a somatória dos "+N+" primeiros números...");
    if (N == 0)
      {
      System.out.println("Não precisamos calcular a somatória dos 0 primeiros "+
                         "números.");
      return 0; // a somatória dos primeiros 0 números inteiros é 0 - não
                // existe a necessidade de quebrar o problema em pedaços
                // menores.
      }
    else
      {
      long sum = somatória(N-1);
      System.out.println("O resultado da somatória dos "+N+" primeiros números é "+
                         N+"+"+sum+" = "+(N+sum));
      return N + sum; // Se N não for igual a zero, retorne N mais a
      // somatória dos (N-1) primeiros números inteiros.
      }
    } // fim do método somatória
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
55
56   }  // fim da classe SomatoriaRecursiva
```

O programa acima, quando executado com o valor 6 entrado pelo teclado, resulta na listagem a seguir (onde as linhas foram indentadas para facilitar a leitura, e erros de concordância nos resultados foram mantidos para que o programa ficasse mais simples):

```
1  Vamos calcular a somatória dos 6 primeiros números...
2    Vamos calcular a somatória dos 5 primeiros números...
3      Vamos calcular a somatória dos 4 primeiros números...
4        Vamos calcular a somatória dos 3 primeiros números...
5          Vamos calcular a somatória dos 2 primeiros números...
6            Vamos calcular a somatória dos 1 primeiros números...
7              Vamos calcular a somatória dos 0 primeiros números...
8                Não precisamos calcular a somatória dos 0 primeiros números.
9            O resultado da somatória dos 1 primeiros números é 1+0 = 1
10         O resultado da somatória dos 2 primeiros números é 2+1 = 3
11       O resultado da somatória dos 3 primeiros números é 3+3 = 6
12     O resultado da somatória dos 4 primeiros números é 4+6 = 10
13   O resultado da somatória dos 5 primeiros números é 5+10 = 15
14 O resultado da somatória dos 6 primeiros números é 6+15 = 21
```

As seis últimas linhas da listagem acima mostram que a somatória de todos os valores até um número é igual à somatória do número mas a somatória de todos os valores anteriores à ele.

Outro exemplo de recursão é usado para o cálculo dos valores da série de Fibonacci. Nesta série, consideramos que o primeiro e o segundo elemento são iguais a um, e que o terceiro elemento e posteriores são calculados como a soma dos dois elementos anteriores, ou $f_0 = 1; f_1 = 1; f_n = f_{(n-1)} + f_{(n-2)}$, se $n > 1$;. A aplicação na listagem 7.10 demonstra a implementação recursiva do cálculo da série de Fibonacci.

Listagem 7.10: A classe `FibonacciRecursiva`, que demonstra o cálculo da série de Fibonacci recursivamente.

```
1  /**
2   * A classe FibonacciRecursiva, que demonstra o cálculo da somatória dos primeiros
3   * N números inteiros através da chamada recursiva de um método estático.
4   */
5  class FibonacciRecursiva
6    {
7    /**
8     * O método main permite a execução desta classe. Este método calcula o N-ésimo
9     * elemento da série de Fibonacci, onde N é entrado pelo teclado.
10    * @param argumentos os argumentos que podem ser passados para o método via linha
11    *        de comando, mas que neste caso serão ignorados.
12    */
13   public static void main(String[] argumentos)
14     {
15     System.out.println("Este programa calcula o N-ésimo elemento da série de "+
16                        "Fibonacci.");
17     System.out.print("Entre o valor de N. Note que o primeiro valor é zero:");
18     int n = Keyboard.readInt();
19     long resultado = fibonacci(n);
20     System.out.println("O "+n+"-ésimo elemento da série de Fibonacci é "+
21                        resultado+".");
22     } // fim do método main
23
24   /**
25    * O método fibonacci calcula o N-ésimo elemento da série de Fibonacci de maneira
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
26    * recursiva. O valor de N é passado como argumento. Se N for zero ou um, o método
27    * retornará um, senão o método retornará a soma dos elementos N-1 e N-2 da série,
28    * chamando novamente o método.
29    * @param N o índice do elemento da série de Fibonacci
30    * @return o valor do elemento da série de Fibonacci
31    */
32    static long fibonacci(int N)
33      {
34      if (N == 0) return 1;
35      else if (N == 1) return 1;
36      else return fibonacci(N-1)+fibonacci(N-2);
37      } // fim do método fibonacci
38
39    }  // fim da classe FibonacciRecursiva
```

Apesar da solução para o cálculo da série de Fibonacci ser simples e sua implementação refletir diretamente a sua definição matemática, ela tem um problema de performance: vários valores da série são calculados várias vezes, e o método é chamado recursivamente mais vezes do que o estritamente necessário. A figura 7.1 ilustra o que ocorre com as chamadas recursivas ao método quando $N = 6$, onde cada caixa corresponde a uma chamada ao método e as setas mostra que métodos com argumentos diferentes serão chamados recursivamente - de acordo com a figura, o método é chamado cinco vezes para calcular o valor de Fibonacci para $n = 2$, o que é certamente dispendioso ! Uma abordagem não-recursiva provavelmente seria mais efetiva neste caso.

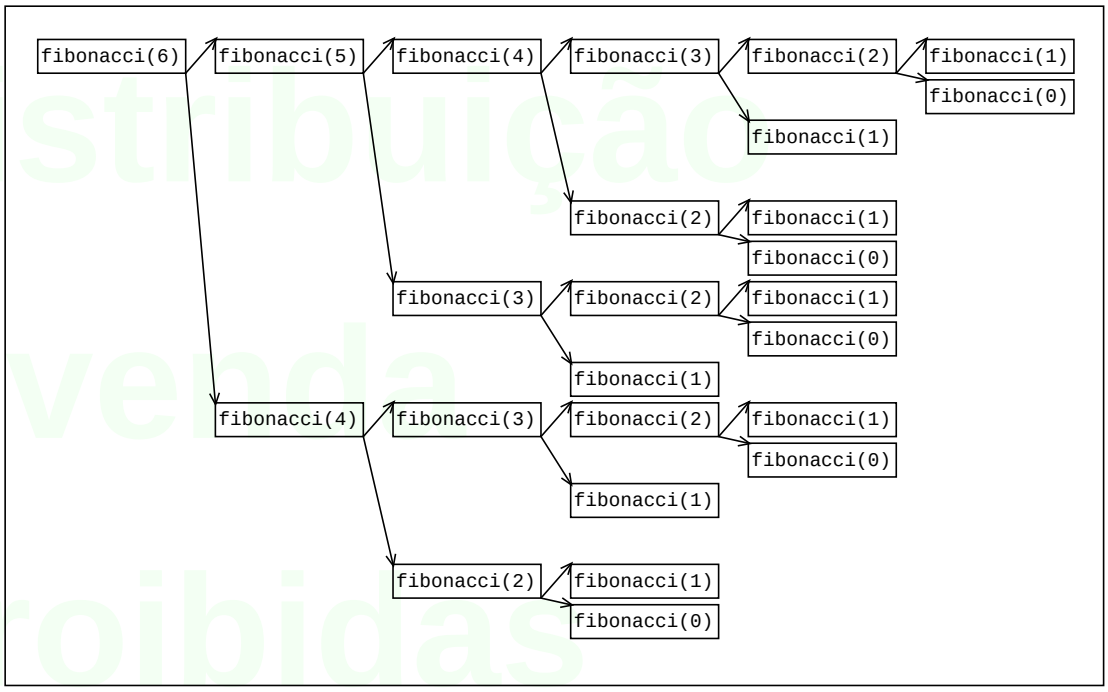



Figura 7.1: Chamadas recursivas ao método `fibonacci`

Vale a pena notar que o uso de recursão oferece soluções elegantes para alguns problemas, mas estas soluções não são necessariamente mais velozes, econômicas (em relação ao uso de memória) ou mesmo, em alguns casos, mais claras. Se houver uma solução para o algoritmo que use laços sem recursão e se velocidade ou uso de memória forem críticos, a solução sem recursão deve ser usada.

Em outros capítulos veremos soluções recursivas para problemas que serão mais simples ou eficientes que soluções não-recursivas.

## 7.3 Introdução à otimização de laços

*otimização de laços*

Otimização é o processo de ajustar partes do código para que um programa seja executado mais rapidamente ou usando menos memória. Otimização de código é possível porque quase sempre existe mais de uma maneira de se implementar um algoritmo usando as ferramentas da linguagem de programação escolhida, e uma das maneiras pode ser mais veloz ou econômica que as outras.

Otimização também tem um preço: código otimizado para a redução de uso de memória ou tempo de processamento pode se tornar mais obscuro, dificultando a sua criação e futura manutenção. Algums compiladores são capazes de otimizar certas partes de código automaticamente.

Otimização não é sempre obrigatória: em especial, para o aprendizado de uma linguagem de programação e para a criação de programas que sejam claros (e que possam no futuro ser compreendidos pelo próprio autor do programa e por outros), os benefícios obtidos pela otimização devem ser contrapostos às dificuldades de implementação.

Nesta seção veremos algumas sugestões sobre como otimizar programas para que sejam executados mais rapidamente. As diferenças entre o tempo de processamento de código otimizado para código não-otimizado são mais claras quando o código é executado múltiplas vezes: a otimização de laços é a que dá resultados mais marcantes.

Para algumas das sugestões apresentadas, uma classe executável que implementa uma solução não-otimizada e uma ou mais soluções otimizadas são apresentadas. Nestas classes, marcamos o tempo de execução de um trecho do código com uma técnica simples: antes de iniciar o trecho em questão anotamos a hora, e repetimos a operação ao fim do trecho. O tempo de execução é obtido subtraindo a hora final da hora inicial. Java permite que a hora atual seja obtida, com precisão e de forma que possa ser usada em cálculos, com o método estático `currentTimeMillis` da classe `System`. Este método retorna o número de milissegundos passados desde a meia-noite de primeiro de janeiro de 1970 como um valor do tipo `long`. Para testar o tempo de execução dos trechos otimizados e não-otimizados dois computadores foram usados: um K6II 400Mhz e um Pentium III 866Mhz, ambos executando o sistema operacional Linux.

*remoção de expressões invariantes*

Sugestão 1: **Evite o recálculo de expressões invariantes em laços.** Expressões invariantes em laços são aquelas que são recalculadas a cada iteração, mas sempre com o mesmo resultado. A listagem 7.11 mostra uma classe executável que calcula o ângulo em radianos correspondente a um ângulo em graus, com um laço que efetua o cálculo para todos os ângulos entre 0 e 360 graus com precisão de centésimo de milésimo de grau. Os valores são convertidos de graus para radianos usando-se um fator de conversão, ou seja, dividindo-os por 180 e multiplicando-os por $\pi$ (cujo valor pode ser obtido na classe `Math`, com o campo constante `PI`). Os valores são somente calculados, não sendo usados posteriormente.

Listagem 7.11: A classe `OtimizacaoRemocaoDeInvariantesEmLacos`, que demonstra otimização de código eliminando partes invariantes de laços.

```
1 /**
2  * A classe OtimizacaoRemocaoDeInvariantesEmLacos, que demonstra como códigos podem
3  * ser otimizados através da redução de código invariante de dentro de laços.
4  */
5 class OtimizacaoRemocaoDeInvariantesEmLacos
6   {
7  /**
```

```
8   * O método main permite a execução desta classe. Este método calculará duas vezes
9   * uma somatória de valores, sendo que em uma delas a parte invariante do laço é
10  * calculada fora deste.
11  * @param argumentos os argumentos que podem ser passados para o método via linha
12  *        de comando, mas que neste caso serão ignorados.
13  */
14  public static void main(String[] argumentos)
15    {
16    long antes,depois; // variáveis que receberão o tempo corrente em milisegundos
17    // Marcamos o tempo presente
18    antes = System.currentTimeMillis();
19    // Fazemos 36000001 iterações com um laço (aparentemente deveriam ser 36000000
20    // mas como o contador é do tipo double, existem erros de arredondamento). A
21    // precisão do contador no for é dada por uma fórmula simples.
22    double fatorDePrecisão = 100000.;
23    for(double ânguloEmGraus=0;ânguloEmGraus<360;ânguloEmGraus+=1/fatorDePrecisão)
24      {
25      // Conversão de graus para radianos, usando uma fórmula simples. Parte desta
26      // fórmula é invariante durante o laço, mas é recalculada em cada iteração.
27      double convertido = ânguloEmGraus*Math.PI/180.;
28      }
29    // Marcamos de novo o tempo presente
30    depois = System.currentTimeMillis();
31    System.out.println("Sem otimização: tempo = "+(depois-antes));
32    // Refazemos o cálculo, desta vez com a parte invariante da fórmula calculada
33    // previamente e simplesmente reutilizada dentro do laço.
34    // Marcamos o tempo presente
35    antes = System.currentTimeMillis();
36    // Calculamos o fator de conversão e o armazenamos em uma variável
37    double fatorDeConversão = Math.PI/180.;
38    // Fazemos 36000001 iterações com um laço
39    for(double ânguloEmGraus=0;ânguloEmGraus<360;ânguloEmGraus+=1/fatorDePrecisão)
40      {
41      // Conversão de graus para radianos, usando uma fórmula simples.
42      double convertido = ânguloEmGraus*fatorDeConversão;
43      }
44    // Marcamos de novo o tempo presente
45    depois = System.currentTimeMillis();
46    System.out.println("Com otimização: tempo = "+(depois-antes));
47    // Refazemos o cálculo, desta vez com as partes invariantes da fórmula e do
48    // incremento do laço calculados previamente e reutilizados dentro do laço.
49    // Marcamos o tempo presente
50    antes = System.currentTimeMillis();
51    // Calculamos o incremento uma só vez
52    double incremento = 1/fatorDePrecisão;
53    // Fazemos 36000001 iterações com um laço
54    for(double ânguloEmGraus=0;ânguloEmGraus<360;ânguloEmGraus+=incremento)
55      {
56      // Conversão de graus para radianos, usando uma fórmula simples.
57      double convertido = ânguloEmGraus*fatorDeConversão;
58      }
59    // Marcamos de novo o tempo presente
60    depois = System.currentTimeMillis();
61    System.out.println("Com mais otimização: tempo = "+(depois-antes));
62    } // fim do método main
63
64  } // fim da classe OtimizacaoRemocaoDeInvariantesEmLacos
```

Alguns pontos interessantes da listagem 7.11 são:

- No laço não-otimizado (linhas 24 a 29) o valor a ser incrementado ao contador do laço é recalculado e o ângulo é multiplicado pelo fator de conversão a cada iteração do laço. Ambas as expressões são invariantes, isto é, independentes do valor da variável de controle no laço. O tempo médio de execução deste laço foi de 49.5 segundos no computador com processador K6 e 2.7 segundos no computador com processador Pentium III.
- No primeiro laço otimizado (linhas 40 a 44) o valor a ser incrementado ao contador do

laço (que é invariante) ainda é calculado dentro do laço, sendo executado a cada iteração deste. O fator de conversão foi calculado e armazenado em uma variável para uso no laço. O tempo médio de execução deste laço foi de 40.2 segundos no computador com processador K6 e 1.3 segundos no computador com processador Pentium III.

- No segundo laço otimizado (linhas 55 a 59) o valor a ser incrementado ao contador do laço e o fator de conversão são calculados antes do laço e armazenados em variáveis. O tempo médio de execução deste laço foi de 32.5 segundos no computador com processador K6 e 0.6 segundos no computador com processador Pentium III.

*uso de variáveis de controle adequadas*

Sugestão 2: **Para laços que são contadores simples, use o tipo `int` para a variável de controle.** Quando laços forem usados exclusivamente para repetição de um procedimento ou como contadores, use preferencialmente o tipo `int` para a variável de controle. Laços com variáveis de controle dos tipos `long`, `float` e `double` executarão mais lentamente do que laços com variáveis de controle do tipo `int`.

Vale a pena notar que em alguns casos laços controlados por variáveis de ponto flutuante são desejáveis: nos laços da listagem 7.11 o cálculo deve ser feito usando uma variável de controle de ponto flutuante, que será usada em um cálculo dentro do laço. Para tentar otimizar este laço podemos usar um pequeno truque que consiste em usar uma variável de controle do tipo `int` como contador e uma variável auxiliar de ponto flutuante que pode ser calculada a partir da variável de controle. Desta forma, o laço entre as linhas 55 e 60 da listagem 7.11 poderia ser escrito como:

Listagem 7.12: Trecho de código que demonstra o uso de variáveis auxiliares de ponto flutuante para evitar variáveis de controle de ponto flutuante em laços.

```
1 for(int ânguloEmGraus=0;ânguloEmGraus<36000000;ânguloEmGraus++)
2   {
3   // Conversão de graus para radianos, usando uma fórmula simples.
4   double convertido = (ânguloEmGraus/fatorDePrecisão)*fatorDeConversão;
5   }
```

O laço mostrado no trecho de código na listagem 7.12 será executado mais rapidamente do que o entre as linhas 55 e 60 da listagem 7.11, mesmo com o cálculo adicional da divisão do ângulo em graus pelo fator de precisão, mas à custa de alguma clareza no programa (à primeira vista o laço aparentemente contará até 36 milhões de graus !).

*substituição de laços curtos*

Sugestão 3: **Evite laços curtos, substituindo-os onde for possível.** O processamento de laços curtos pode ser substituído por código sequencial onde for possível, evitando a inicialização, modificação e comparação da variável de controle - técnica conhecida como desenrolamento de laços. O programa na listagem 7.13 mostra como podemos calcular a somatória

$$\sum_{i=0}^{i=9999999} \frac{1}{\sum_{j=i}^{j=i+4} j}$$

de três maneiras, com tempos de processamento diferentes.

Listagem 7.13: A classe `OtimizacaoDesenrolamentoDeLacos`, que demonstra otimização de código eliminando pequenos laços onde for possível.

```
/**
 * A classe OtimizacaoDesenrolamentoDeLacos, que demonstra como códigos podem ser
 * otimizados através do uso de código sequencial ao invés de pequenos laços.
 */
class OtimizacaoDesenrolamentoDeLacos
  {
 /**
  * O método main permite a execução desta classe. Este método calculará três vezes
  * uma somatória de valores, que usa como termo uma mini-seqüência de valores que
  * no primeiro caso é calculada com um laço, no segundo como uma somatória direta e
  * no terceiro como uma operação mais simples com menos termos.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    long antes,depois; // variáveis que receberão o tempo corrente em milisegundos
    double soma = 0; // variável que receberá o valor de uma somatória
    // Marcamos o tempo presente
    antes = System.currentTimeMillis();
    // Executamos 10000000 iterações
    for(int contador=0;contador<10000000;contador++)
      {
      // Precisamos somar cinco valores sequenciais iniciando com o contador:
      // sem otimizar, usamos um laço.
      double somaInterna = 0;
      for(int contadorInterno=contador;contadorInterno<contador+5;contadorInterno++)
        somaInterna += contadorInterno;
      // Calculamos o termo da seqüência e o somamos
      soma += 1./somaInterna;
      }
    // Marcamos de novo o tempo presente
    depois = System.currentTimeMillis();
    System.out.println("Sem otimização: tempo = "+(depois-antes));
    // Refazemos o cálculo, desta vez substituindo o laço interno por uma operação
    // direta com o mesmo resultado.
    soma = 0; // reinicializamos a somatória
    // Marcamos o tempo presente
    antes = System.currentTimeMillis();
    // Executamos 10000000 iterações
    for(int contador=0;contador<10000000;contador++)
      {
      // Somamos diretamente os cinco valores sequenciais e armazenamos em uma
      // variável, ao invés de usar um laço.
      double somaInterna = (contador+contador+1+contador+2+contador+3+contador+4);
      soma += 1./somaInterna;
      }
    // Marcamos de novo o tempo presente
    depois = System.currentTimeMillis();
    System.out.println("Com otimização: tempo = "+(depois-antes));
    // Refazemos o cálculo, desta vez escrevendo a operação com menos termos
    soma = 0; // reinicializamos a somatória
    // Marcamos o tempo presente
    antes = System.currentTimeMillis();
    for(int contador=0;contador<10000000;contador++)
      {
      // Usamos uma expressão mais concisa ao invés da soma com vários termos
      double somaInterna = 10+contador*5;
      soma += 1./somaInterna;
      }
    // Marcamos de novo o tempo presente
    depois = System.currentTimeMillis();
    System.out.println("Com mais otimização: tempo = "+(depois-antes));
    } // fim do método main

  } // fim da classe OtimizacaoDesenrolamentoDeLacos
```

Os resultados de processamento e comentários sobre os laços na listagem 7.11 são:

- O primeiro laço, entre as linhas 23 e 32, não faz nenhuma otimização, calculando internamente a segunda somatória usando outro laço curto. Os tempos de processamento deste laço foram 37.1 segundos para o K6 e 14.5 segundos para o Pentium III.
- O segundo laço, entre as linhas 42 e 48, calcula a somatória diretamente usando uma variável auxiliar, evitando o laço interno, e executa em 13.1 segundos para o K6 e 8.1 segundos para o Pentium III.
- O terceiro laço, entre as linhas 56 e 61, otimiza ainda mais o processamento usando uma fórmula mais simples para calcular o valor da somatória interna, diminuindo o número de operações que devem ser feitas. Os tempos de processamento foram 8.9 segundos para o K6 e 4 segundos para o Pentium III.

*eliminação de chamadas desnecessárias a métodos*

Sugestão 4: **Evite chamadas a métodos quando existirem maneiras de efetuar operações diretamente.** Toda chamada a um método tem um custo de processamento, que não faz muita diferença quando o método é chamado poucas vezes mas que pode tomar boa parte do tempo de processamento quando o método é chamado repetidamente de dentro de um laço. A classe executável na listagem 7.14 demonstra laços que calculam a somatória

$$\sum_{i=0}^{i=9999999} \frac{i-1}{i+1}$$

usando métodos para efetuar alguns cálculos e sem usar métodos, efetuando os cálculos diretamente.

Listagem 7.14: A classe `OtimizacaoRemocaoDeChamadasAMetodos`, que demonstra otimização de código eliminando chamadas desnecessárias à métodos..

```
1  /**
2   * A classe OtimizacaoRemocaoDeChamadasAMetodos, que demonstra como códigos podem ser
3   * otimizados através da remoção de código invariante de dentro de laços.
4   */
5  class OtimizacaoRemocaoDeChamadasAMetodos
6    {
7   /**
8    * O método main permite a execução desta classe. Este método calculará duas vezes
9    * uma somatória de valores, sendo que em uma delas métodos são usados para o
10   * cálculo enquanto que a outra implementa o cálculo diretamente.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     long antes,depois; // variáveis que receberão o tempo corrente em milisegundos
17     double soma; // variável para receber o valor da somatória
18     // Marcamos o tempo presente
19     antes = System.currentTimeMillis();
20     // Fazemos 10000000 iterações com um laço
21     soma = 0;
22     for(int contador=0;contador<10000000;contador++)
23       {
24       // Fazemos a somatória dos termos usando duas chamadas a métodos.
25       soma += adiciona(contador,-1)/adiciona(contador,+1);
26       }
27     // Marcamos de novo o tempo presente
28     depois = System.currentTimeMillis();
29     System.out.println("Sem otimização: tempo = "+(depois-antes));
30     // Refazemos o cálculo, desta vez efetuando os cálculos diretamente ao invés
31     // de usar um método.
32     // Marcamos o tempo presente
33     antes = System.currentTimeMillis();
34     // Fazemos 10000000 iterações com um laço
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
35      soma = 0;
36      for(int contador=0;contador<10000000;contador++)
37        {
38        // Fazemos a somatória dos termos usando duas chamadas a métodos.
39        soma += (contador-1.)/(contador+1.);
40        }
41      // Marcamos de novo o tempo presente
42      depois = System.currentTimeMillis();
43      System.out.println("Com otimização: tempo = "+(depois-antes));
44      } // fim do método main
45
46    /**
47     * O método adiciona simplesmente adiciona os valores passados como argumentos e
48     * retorna o resultado. Este tipo de método pode ser facilmente substituido para
49     * evitar que deva ser chamado.
50     * @param valor o valor base que será adicionado
51     * @param delta o valor a ser adicionado ao valor base
52     * @return a soma dos argumentos à este método
53     */
54    public static double adiciona(double valor,double delta)
55      {
56      return valor+delta; // simplesmente adiciona os argumentos e retorna a soma
57      } // fim do método adiciona
58
59    } // fim da classe OtimizacaoRemocaoDeChamadasAMetodos
```

Os resultados de processamento e comentários sobre os laços na listagem 7.14 são:

- O laço entre as linhas 23 e 27 usa duas chamadas a métodos para calcular os termos da somatória, e seu tempo médio de execução é de 20.5 segundos no K6 e de 6 segundos no Pentium III.
- O laço entre as linhas 37 e 41 dispensa a chamada ao método, implementando diretamente o cálculo dos termos da somatória, e seu tempo médio de execução é de 10.5 segundos no K6 e de 3.6 segundos no Pentium III.

Vale a pena notar que no exemplo acima a clareza do código não fica muito comprometida com a dispensa do método pois o corpo deste é bem simples. Se o cálculo efetuado por um método for mais complexo, poderá valer a pena sacrificar parte da performance para manter a clareza do código.

## 7.4 Exercícios do capítulo 7

**Exercício 7.1:** ⋆

Após a execução do trecho de código `int a,b,c; a=2; b=3; c= a++ + b++;`, quais serão os valores das variáveis ?

**A.** a=3, b=4 e c=7.
**B.** a=3, b=4 e c=5.
**C.** a=2, b=3 e c=7.
**D.** a=2, b=3 e c=5.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.2:** ⋆

Após a execução do trecho de código `long x,y,z; x=0; y=12; z= ++x + ++y;`, quais serão os valores das variáveis ?

**A.** `x=0`, `y=12` e `z=12`.
**B.** `x=0`, `y=12` e `z=14`.
**C.** `x=1`, `y=13` e `z=12`.
**D.** `x=1`, `y=13` e `z=14`.

**Exercício 7.3:** ⋆

Após a execução do trecho de código `double r,s; r=9; s= -r + r++;`, quais serão os valores das variáveis ?

**A.** `r=9` e `s=18`.
**B.** `r=8` e `s=18`.
**C.** `r=9` e `s=16`.
**D.** `r=8` e `s=16`.

**Exercício 7.4:** ⋆

Após a execução do trecho de código `float n,o,p; p=0; n=3; o=2; p -= n+=2; p -= o+=3;`, quais serão os valores das variáveis ?

**A.** `n=3`, `o=2` e `p=-3`.
**B.** `n=3`, `o=2` e `p=-10`.
**C.** `n=5`, `o=5` e `p=-3`.
**D.** `n=5`, `o=5` e `p=-10`.

**Exercício 7.5:** ⋆

Durante a execução do seguinte trecho de código

```
for(System.out.print("Início,");true;System.out.print("Fim."))
  {
  System.out.print("Executando,");
  }
```

o que será mostrado no terminal ?

**A.** `Início,Executando,Executando,Executando,...` (repetido infinitamente)
**B.** `Início,Executando,Fim.Executando,Fim.Executando,Fim...` (repetido infinitamente)
**C.** `Início,Executando,Fim.` (sem repetição)
**D.** Um erro de execução ocorrerá.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.6:** ⋆

Durante a execução do seguinte trecho de código

```
for(System.out.print("Início,");true;System.out.print("Fim."))
  {
  System.out.print("Executando,");
  }
```

o que será mostrado no terminal ?

**A.** `Início,Executando,` (sem repetição)
**B.** `Início,` (sem repetição)
**C.** `Início,Executando,Fim.` (sem repetição)
**D.** Um erro de execução ocorrerá.

**Exercício 7.7:** ⋆

Por que um bloco `while` iniciado por `while(true)` pode ser útil enquanto um bloco iniciado por `while(false)` certamente será inútil ? Explique.

**Exercício 7.8:** ⋆

O que acontecerá se removermos a linha 43 da listagem 7.3 ? Explique.

**Exercício 7.9:** ⋆

O que aconteceria se o valor inicial da variável `fatorial` no método `fatorial` da classe `ProbabilidadeBasica` (listagem 7.7) fosse inicializado com zero ?

**Exercício 7.10:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1 class ContadorComWhile
2   {
3   public static void main(String[] argumentos)
4     {
5     int contador = 0;
6     while(contador != 100)
7       contador = contador+3;
8     }
9   } // fim da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.11:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class ContadorComWhile
2    {
3    public static void main(String[] argumentos)
4      {
5      double valor = 100;
6      while(valor < 100)
7        {
8        valor /= 2;
9        }
10     }
11   } // fim da classe
```

**Exercício 7.12:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class ContadorComFor
2    {
3    public static void main(String[] argumentos)
4      {
5      double a,b=1;
6      for(a=0;a<1000;b++)
7        {
8        System.out.println(a+" "+b);
9        }
10     }
11   } // fim da classe
```

**Exercício 7.13:** ⋆

Reescreva os laços `while` na classe `DemoWhile` (listagem 7.1) usando laços `for`.

**Exercício 7.14:** ⋆

Reescreva os laços `for` na classe `DemoFor` (listagem 7.6) usando laços `while` ou `do-while`.

**Exercício 7.15:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class ContadorComFor
2    {
3    public static void main(String[] argumentos)
4      {
5      for(int l=0;l<25;l++)
6        {
7        for(int l=0;l<80;l++)
8          System.out.print("*");
9        System.out.println();
10       }
11     }
12   } // fim da classe
```

**Exercício 7.16:** ⋆

Explique por que o uso de recursão deve ser implementado através de métodos.

**Exercício 7.17:** ⋆⋆

Considere duas variáveis `X` e `Y` que possam assumir valores entre -100 e 100. Escreva um programa em Java que imprima todos os valores de `X` e `Y` para os quais a soma `X + Y` seja igual a 100 ou igual a -100.

**Exercício 7.18:** ⋆⋆

Escreva uma classe `Loteria` que tenha métodos estáticos para imprimir versões aproximadas dos cartões da Mega-Sena e LotomMania (com somente os números, respeitando o número de linhas e a distribuição dos números nas linhas).

**Exercício 7.19:** ⋆⋆

Usando a classe `JogoDeAdivinhacao` (listagem 7.4) como base, escreva uma classe `SenhaDeCaixaEletronico` com um método que peça ao usuário entrar com uma senha numérica, e se em três tentativas o usuário não acertar a senha, imprima uma mensagem de erro. Escreva também uma aplicação que demonstre a classe.

**Exercício 7.20:** ⋆⋆

Reescreva o método `tenta` da classe `JogoDeAdivinhacao` (listagem 7.4) de forma que o `if` com `break` sejam eliminados, e que as duas condições de término do laço (número correto ou fim do número de tentativas) seja tratado pela instrução `while` do bloco `do-while`.

**Exercício 7.21:** ⋆⋆

Escreva um programa em Java que imprima a série de Fibonacci até o $N$-ésimo elemento, sem usar recursão. O número de elementos $N$ pode ser lido do teclado.

**Exercício 7.22:** ⋆⋆

O laço das linhas 55 a 59 da classe `OtimizacaoRemocaoDeInvariantesEmLacos` (listagem 7.11) pode ser mais otimizado se uma variável de controle inteira for usada, como mostrado pelo trecho de código da página 143. O laço pode ser ainda mais otimizado se a parte invariante do cálculo `double convertido = (ânguloEmGraus/fatorDePrecisão)*fatorDeConversão;` for retirada do laço. Modifique o laço de forma a fazer esta otimização.

**Exercício 7.23:** ⋆⋆⋆

Modifique as classes `JogoDeAdivinhacao` e `DemoJogoDeAdivinhacao` (listagens 7.4 e 7.5) para que o usuário tenha que adivinhar um de dois números escolhidos. Dois números devem ser passados para o construtor, e o método `tenta` deve dizer, quando o usuário entrar um valor pelo teclado, se este valor é maior ou menor do que cada um dos dois valores secretos. Se o usuário acertar um dos dois valores, ele vence o jogo.

**Exercício 7.24:** ⋆⋆⋆

Escreva uma classe `Serie` que encapsule o mecanismo de geração de séries numéricas como as usadas em testes de raciocínio, onde uma pessoa deve deduzir a regra que gerou os números e acertar os próximos números da série. A série deve ser gerada usando três valores: *inicial*, *multiplicador* e *adicional*, de forma que o primeiro número da série será igual a *inicial*, o segundo será calculado como (*inicial* + *adicional*) * *multiplicador*, o terceiro como (*segundo* + *adicional*) * *multiplicador* e assim sucessivamente. Os valores devem ser passados como argumentos para o construtor da classe, e usados por um método `imprime` que recebe, como argumento, o número de termos que serão impressos. Por exemplo, uma aplicação poderia criar três instâncias da classe `Serie` e imprimir, respectivamente, os primeiros 10, 12 e 14 termos, com o trecho de código abaixo:

```
Serie s1 = new Serie(0,-2,2);
s1.imprime(10);
Serie s2 = new Serie(1,2,0);
s2.imprime(12);
Serie s3 = new Serie(1,1,2);
s3.imprime(14);
```

O resultado da execução do trecho de código acima é mostrado abaixo.

```
0 -4 4 -12 20 -44 84 -172 340 -684
1 2 4 8 16 32 64 128 256 512 1024 2048
1 3 5 7 9 11 13 15 17 19 21 23 25 27
```

**Exercício 7.25:** ⋆⋆⋆

Modifique a classe `Serie` (exercício 7.24) para que o método `imprime` receba um argumento inicial do tipo `int` que indica qual dos valores da série deve ser impresso como um asterisco - desta forma fica mais fácil simular testes de raciocínio. Por exemplo, uma aplicação poderia criar uma série como `Serie novaSerie = new Serie(-20,-2,-3);` `novaSerie.imprime(10,5);` , e o resultado da execução deste trecho de código seria `-20 46 -86 178 * 706 -1406 2818 -5630 11266` (o quinto elemento da série está oculto).
Modifique também o construtor da classe `Serie` para que ele mostre uma mensagem de erro caso o valor passado para o multiplicador seja zero (para evitar que a série seja composta somente de zeros).

**Exercício 7.26:** ⋆⋆⋆

Escreva uma aplicação em Java que calcule o máximo divisor comum de dois números. O algoritmo de Euclides para o cálculo do máximo divisor comum entre dois números positivos $M$ e $N$ calcula $MDC(M,N)$ como:

- Se $N > M$, retorne $MDC(N,M)$.
- Se $N = 0$, retorne $M$.
- Senão, retorne $MDC(N, M\%N)$ (onde % é o operador módulo, que retorna o resto da divisão).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.27:** ⋆⋆⋆

A raiz quadrada de um número pode ser encontrada usando-se um algoritmo recursivo, que usa como entrada três valores: $N$ que é o número do qual queremos calcular a raiz quadrada; $A$ que é uma aproximação inicial da raiz quadrada, e $E$ que é o máximo de erro que pode ser admitido no cálculo. O algoritmo é como segue:

- Se o valor absoluto de $A^2 - N$ for menor do que $E$, retorne $A$,
- Senão, faça $A = (A^2 + N)/(2A)$ e execute novamente o algoritmo.

O valor absoluto pode ser calculado com o método `Math.abs`.

**Exercício 7.28:** ⋆⋆⋆

Escreva um programa em Java que calcule a função de Ackermann. A função de Ackermann ($A$) é calculada como:

- Se $x == 0$, $A(x,y) = y+1$.
- Se $y == 0$, $A(x,y) = A(x-1,1)$.
- Senão, $A(x,y) = A(x-1,A(x,y-1))$.

**Evite tentar calcular o valor desta função para valores grandes de $x$ e/ou $y$ - o cálculo de $A(3,4)$ gerou mais de 10.000 chamadas recursivas.**

**Exercício 7.29:** ⋆⋆⋆⋆

Implemente uma solução recursiva para a geração de séries de números do exercício 7.24, de forma que o método `imprime` chame um outro método, que por sua vez será chamado recursivamente, passando para este outro método os argumentos necessários para a geração da série. Como será feito o controle de parada da recursão ?

## 7.5 Exercícios do capítulo 7 que envolvem séries matemáticas

A criação de classes e programas que calculam séries matemáticas requer que o programador escreva laços com condicionais de diferentes graus de complexidade. Para reforçar os conceitos apresentados neste capítulo e no anterior, vários exercícios desta natureza são apresentados.

Todos os exercícios sobre séries matemáticas neste capítulo foram retirados do livro *Manual de Fórmulas e Tabelas Matemáticas*, de Murray R. Spiegel, da coleção Schaum da editora McGraw-Hill (1973), do artigo *On the rapid computation of various polylogarithmic constants* de David Bailey, Peter Borwein e Simon Plouffe, da página da Mathsoft (http://www.mathsoft.com) e da página *Mathematical Constants and Computation*, de Xavier Gourdon e Pascal Sebah (http://numbers.computation.free.fr/Constants/constants.html).

**Exercício 7.30:** ⋆⋆

O valor de $x^y$ pode ser calculado como sendo $x$ multiplicado por si mesmo $y$ vezes (se $y$ for inteiro). Escreva uma classe `SeriesMatematicas` que contenha o método estático `elevadoA` que receba como argumentos os valores $x$ e $y$ e calcule e retorne $x^y$.

**Exercício 7.31:** ⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piQuadradoSobre6` que calcule a série $\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}+\frac{1}{5^2}+\cdots$. Evidentemente a série não poderá ser calculada infinitamente, devendo parar depois de $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^2}{6}$. Um recurso interessante é calcular o valor esperado da série com a expressão `(Math.PI * Math.PI)/6` e ver quão diferente deste valor esperado é o calculado pela série. A diferença pode ser mostrada passo a passo, para ver se a série está convergindo. Escreva também um programa que demonstre o método `piQuadradoSobre6`. Veja também o exercício 7.30.

**Exercício 7.32:** ⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piQuartaSobre90` que calcule a série $\frac{1}{1^4}+\frac{1}{2^4}+\frac{1}{3^4}+\frac{1}{4^4}+\frac{1}{5^4}+\cdots$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^4}{90}$. *Dica:* $X^4$ pode ser calculado simplesmente como $X*X*X*X$. Veja também o exercício 7.30.

**Exercício 7.33:** ⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piSextaSobre945` que calcule a série $\frac{1}{1^6}+\frac{1}{2^6}+\frac{1}{3^6}+\frac{1}{4^6}+\frac{1}{5^6}+\cdots$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^6}{945}$. *Dica:* $X^6$ pode ser calculado simplesmente como $X*X*X*X*X*X$. Veja também os exercícios 7.30 e 7.31.

**Exercício 7.34:** ⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piQuadradoSobre8` que calcule a série $\frac{1}{1^2}+\frac{1}{3^2}+\frac{1}{5^2}+\frac{1}{7^2}+\frac{1}{9^2}+\cdots$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^2}{8}$. Veja também o exercício 7.30.

**Exercício 7.35:** ⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piQuartaSobre96` que calcule a série $\frac{1}{1^4}+\frac{1}{3^4}+\frac{1}{5^4}+\frac{1}{7^4}+\frac{1}{9^4}+\cdots$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^4}{96}$. Veja também os exercícios 7.30 e 7.31.

**Exercício 7.36:** ⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piSextaSobre960` que calcule a série $\frac{1}{1^6}+\frac{1}{3^6}+\frac{1}{5^6}+\frac{1}{7^6}+\frac{1}{9^6}+\cdots$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^6}{960}$. Veja também os exercícios 7.30 e 7.31.

**Exercício 7.37:** ⋆⋆

Escreva um programa em Java que calcule a série $\frac{1}{1\times3}+\frac{1}{3\times5}+\frac{1}{5\times7}+\frac{1}{7\times9}+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $1/2$. Veja também o exercício 7.30.

**Exercício 7.38:** ⋆⋆

Escreva um programa em Java que calcule a série $\frac{1}{1\times3}+\frac{1}{2\times4}+\frac{1}{3\times5}+\frac{1}{4\times6}+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $3/4$. Veja também o exercício 7.30.

**Exercício 7.39:** ⋆⋆

Escreva uma versão recursiva do método `elevadoA` (exercício 7.30) para a classe `SeriesMatematicas`.

**Exercício 7.40:** ⋆⋆⋆

Escreva um programa em Java que calcule a série $\frac{1}{1}-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\frac{1}{5}-\frac{1}{6}+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual ao logaritmo com base natural de 2 ($ln(2) = 0.6931471805599453$). Que mecanismo poderá ser usado para efetuar a troca de sinais em cada passo da série ? Veja também o exercício 7.30.

**Exercício 7.41:** ⋆⋆⋆

Escreva um programa em Java que calcule a série $\frac{1}{1}+\frac{1}{3}-\frac{1}{2}+\frac{1}{5}+\frac{1}{7}-\frac{1}{4}+\frac{1}{9}+\frac{1}{11}-\frac{1}{6}+\frac{1}{13}+\frac{1}{15}-\frac{1}{8}+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual ao logaritmo com base natural de 2 ($ln(2) = 0.6931471805599453$) multiplicado por $3/2$. Que mecanismo poderá ser usado para efetuar a troca de sinais em cada passo da série ? Veja que a série se repete com grupos de dois termos positivos de denominador ímpar com um termo negativo de denominador par. *Dica:* pode ser mais simples calcular vários termos da série em cada iteração do laço do que criar uma lógica que determine corretamente o sinal de cada termo. Veja também o exercício 7.30.

**Exercício 7.42:** ⋆⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `piQuadradoSobre12` que calcule a série $\frac{1}{1^2}-\frac{1}{2^2}+\frac{1}{3^2}-\frac{1}{4^2}+\frac{1}{5^2}-\cdots.$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^2}{12}$. Veja também o exercício 7.31.

**Exercício 7.43:** ⋆⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `setePiQuartaSobre720` que calcule a série $\frac{1}{1^4}-\frac{1}{2^4}+\frac{1}{3^4}-\frac{1}{4^4}+\frac{1}{5^4}-\cdots.$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{7\pi^4}{720}$. Veja também o exercício 7.31.

**Exercício 7.44:** ⋆⋆⋆

Escreva na classe `SeriesMatematicas` o método estático `trintaEUmPiSextaSobre30240` que calcule a série $\frac{1}{1^6}-\frac{1}{2^6}+\frac{1}{3^6}-\frac{1}{4^6}+\frac{1}{5^6}-\cdots.$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{31\pi^6}{30240}$. Veja também o exercício 7.31.

**Exercício 7.45:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `piCuboSobre32` que calcule a série $\frac{1}{1^3}+\frac{1}{3^3}-\frac{1}{5^3}+\frac{1}{7^3}-\frac{1}{9^3}+\cdots$. com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi^3}{32}$. Veja também o exercício 7.31.

**Exercício 7.46:** ★★★

Escreva um programa em Java que calcule a série $\frac{1}{1}+\frac{1}{3}-\frac{1}{5}-\frac{1}{7}+\frac{1}{9}+\frac{1}{11}-\frac{1}{13}-\frac{1}{15}+\frac{1}{17}+\frac{1}{19}-\frac{1}{21}-\frac{1}{23}+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi}{4}\sqrt{2}$. Como fazer a troca de sinais a cada dois passos da série ? Veja que a série é composta de grupos de duas somas e duas subtrações. *Dica:* pode ser mais simples calcular vários termos da série em cada iteração do laço do que criar uma lógica que determine corretamente o sinal de cada termo. Veja também os exercícios 7.30 e 7.41.

**Exercício 7.47:** ★★★

Escreva um programa em Java que calcule a série $1-\frac{1}{2}-\frac{1}{3}+\frac{1}{4}+\frac{1}{5}-\frac{1}{6}-\frac{1}{7}+\frac{1}{8}+\frac{1}{9}-\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi}{4}-\frac{1}{2}ln(2)$. Veja também os exercícios 7.37 e 7.46.

**Exercício 7.48:** ★★★

Escreva um programa em Java que calcule a série $1-\frac{1}{4}-\frac{1}{7}+\frac{1}{10}-\frac{1}{13}+\frac{1}{16}-\frac{1}{19}+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{1}{3}(\frac{\pi}{\sqrt{3}}+ln(2))$.

**Exercício 7.49:** ★★★

Escreva um programa em Java que calcule a série $1-\frac{1}{5}+\frac{1}{9}-\frac{1}{13}+\frac{1}{17}-\frac{1}{21}+\frac{1}{25}-\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{1}{4\sqrt{2}}(\pi+2ln(1+\sqrt{2}))$.

**Exercício 7.50:** ★★★

Escreva um programa em Java que calcule a série $1-\frac{1}{7}+\frac{1}{9}-\frac{1}{15}+\frac{1}{17}-\frac{1}{23}+\frac{1}{25}-\frac{1}{31}+\frac{1}{33}-\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi}{4}(\frac{1+\sqrt{2}}{2})$. Aparentemente não existe uma série para os denominadores dos termos, mas estes podem ser calculados de dois em dois facilmente. Veja também o exercício 7.41.

**Exercício 7.51:** ★★★

Escreva um programa em Java que calcule a série $1-\frac{1}{2}+\frac{1}{4}-\frac{1}{5}+\frac{1}{7}-\frac{1}{8}+\frac{1}{10}-\frac{1}{11}+\frac{1}{13}-\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi\sqrt{3}}{9}$ . Aparentemente não existe uma série para os denominadores dos termos, mas estes podem ser calculados de dois em dois facilmente. Veja também os exercícios 7.41 e 7.50.

**Exercício 7.52:** ★★★

Escreva um programa em Java que calcule a série $(\frac{1}{1}+\frac{1}{2}-\frac{1}{3}+\frac{1}{4}-\frac{1}{5}-\frac{1}{6})+(\frac{1}{8}+\frac{1}{9}-\frac{1}{10}+\frac{1}{11}-\frac{1}{12}-\frac{1}{13})+(\frac{1}{15}+\frac{1}{16}-\frac{1}{17}+\frac{1}{18}-\frac{1}{19}-\frac{1}{20})+\cdots$ com $N$ termos, sendo que valor de $N$ pode ser entrado via teclado. O resultado da série, se calculada infinitamente, será igual a $\frac{\pi\sqrt{7}}{7}$. Aparentemente não existe uma série para os denominadores dos termos, mas estes podem ser calculados de seis em seis facilmente. Veja também os exercícios 7.41 e 7.50.

**Exercício 7.53:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `calculaPi` que calcule a série $4\times\left(\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots\right)$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\pi$. Veja também o exercício 7.30.

**Exercício 7.54:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `calculaPiDeOutraForma` que calcule a série $2\times\frac{2}{1}\times\frac{2}{3}\times\frac{4}{3}\times\frac{4}{5}\times\frac{6}{5}\times\frac{6}{7}\times\cdots$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\pi$. Veja também os exercícios 7.30 e 7.53.

**Exercício 7.55:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `exponencial` que calcule a série $1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\frac{x^4}{4!}+\frac{x^5}{5!}+\cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao valor $e^x$ (onde $e$ é a base dos logaritmos naturais). *Dica:* O fatorial de um número pode ser calculado usando o método `fatorial` na classe `ProbabilidadeBasica` (listagem 7.7). Veja também o exercício 7.30.

**Exercício 7.56:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `seno` que calcule a série $x\times(1-\frac{x^2}{\pi^2})\times(1-\frac{x^2}{4\pi^2})\times(1-\frac{x^2}{9\pi^2})\times(1-\frac{x^2}{16\pi^2})\times\cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao seno do ângulo $x$ em radianos. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`.

**Exercício 7.57:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `senoDeOutraForma` que calcule a série $x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\frac{x^9}{9!}-\cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao seno do ângulo $x$ em radianos. O fatorial de um número pode ser calculado usando o método `fatorial` na classe `ProbabilidadeBasica` (listagem 7.7), e o valor de $x^y$ com o método `elevadoA` (exercício 7.30).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.58:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `senoHiperbólico` que calcule a série $x \times (1+\frac{x^2}{\pi^2}) \times (1+\frac{x^2}{4\pi^2}) \times (1+\frac{x^2}{9\pi^2}) \times (1+\frac{x^2}{16\pi^2}) \times \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao seno hiperbólico do ângulo $x$. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`. Veja também o exercício 7.56.

**Exercício 7.59:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `senoHiperbólicoDeOutraForma` que calcule a série $x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} + \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao seno hiperbólico do ângulo $x$. O fatorial de um número pode ser calculado usando o método `fatorial` na classe `ProbabilidadeBasica` (listagem 7.7), e o valor de $x^y$ com o método `elevadoA` (exercício 7.30). Veja também os exercícios 7.57 e 7.58.

**Exercício 7.60:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `coseno` que calcule a série $(1-\frac{4x^2}{\pi^2}) \times (1-\frac{4x^2}{9\pi^2}) \times (1-\frac{4x^2}{25\pi^2}) \times (1-\frac{4x^2}{49\pi^2}) \times \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao co-seno do ângulo $x$ em radianos. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`.

**Exercício 7.61:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cosenoDeOutraForma` que calcule a série $1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao co-seno do ângulo $x$ em radianos. O fatorial de um número pode ser calculado usando o método `fatorial` na classe `ProbabilidadeBasica` (listagem 7.7), e o valor de $x^y$ com o método `elevadoA` (exercício 7.30).

**Exercício 7.62:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cosenoHiperbólico` que calcule a série $(1+\frac{4x^2}{\pi^2}) \times (1+\frac{4x^2}{9\pi^2}) \times (1+\frac{4x^2}{25\pi^2}) \times (1+\frac{4x^2}{49\pi^2}) \times \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao co-seno hiperbólico do ângulo $x$. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`. Veja também o exercício 7.60.

**Exercício 7.63:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cosenoHiperbólicoDeOutraForma` que calcule a série $1+\frac{x^2}{2!}+\frac{x^4}{4!}+\frac{x^6}{6!}+\frac{x^8}{8!}+\cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao co-seno hiperbólico do ângulo $x$. O fatorial de um número pode ser calculado usando o método `fatorial` na classe `ProbabilidadeBasica` (listagem 7.7), e o valor de $x^y$ com o método `elevadoA` (exercício 7.30). Veja também os exercícios 7.61 e 7.62.

**Exercício 7.64:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `senoDeXSobreX` que calcule a série $cos(\frac{x}{2}) \times cos(\frac{x}{4}) \times cos(\frac{x}{8}) \times cos(\frac{x}{16}) \times cos(\frac{x}{32}) \times \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual a $\frac{sin(x)}{x}$. O co-seno de um número pode ser calculado usando um dos métodos `coseno` ou `cosenoDeOutraForma` (exercícios 7.60 e 7.61).

**Exercício 7.65:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `tangente` que calcule a série $8x \times \left(\frac{1}{\pi^2-4x^2}+\frac{1}{9\pi^2-4x^2}+\frac{1}{25\pi^2-4x^2}+\frac{1}{49\pi^2-4x^2}+\cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à tangente do ângulo $x$ em radianos. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`.

**Exercício 7.66:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `tangenteHiperbólica` que calcule a série $8x \times \left(\frac{1}{\pi^2+4x^2}+\frac{1}{9\pi^2+4x^2}+\frac{1}{25\pi^2+4x^2}+\frac{1}{49\pi^2+4x^2}+\cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à tangente hiperbólica do ângulo $x$. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`. Veja também o exercício 7.65.

**Exercício 7.67:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cotangente` que calcule a série $\frac{1}{x}+2x\left(\frac{1}{x^2-\pi^2}+\frac{1}{x^2-4\pi^2}+\frac{1}{x^2-9\pi^2}+\frac{1}{x^2-16\pi^2}+\cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à co-tangente do ângulo $x$ em radianos. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`.

**Exercício 7.68:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cotangenteHiperbólica` que calcule a série $\frac{1}{x} + 2x\left(\frac{1}{x^2+\pi^2} + \frac{1}{x^2+4\pi^2} + \frac{1}{x^2+9\pi^2} + \frac{1}{x^2+16\pi^2} + \cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à co-tangente hiperbólica do ângulo $x$. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`. Veja também o exercício 7.67.

**Exercício 7.69:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `secante` que calcule a série $4\pi \times \left(\frac{1}{\pi^2-4x^2} - \frac{3}{9\pi^2-4x^2} + \frac{5}{25\pi^2-4x^2} - \frac{7}{49\pi^2-4x^2} + \cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à secante do ângulo $x$ em radianos. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`.

**Exercício 7.70:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `secanteHiperbólica` que calcule a série $4\pi \times \left(\frac{1}{\pi^2+4x^2} - \frac{3}{9\pi^2+4x^2} + \frac{5}{25\pi^2+4x^2} - \frac{7}{49\pi^2+4x^2} + \cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à secante hiperbólica do ângulo $x$. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`. Veja também o exercício 7.69.

**Exercício 7.71:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cosecante` que calcule a série $\frac{1}{x} - 2x\left(\frac{1}{x^2-\pi^2} - \frac{1}{x^2-4\pi^2} - \frac{1}{x^2-9\pi^2} - \frac{1}{x^2-16\pi^2} - \cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à co-secante do ângulo $x$ em radianos. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`.

**Exercício 7.72:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `cosecanteHiperbólica` que calcule a série $\frac{1}{x} - 2x\left(\frac{1}{x^2+\pi^2} - \frac{1}{x^2+4\pi^2} - \frac{1}{x^2+9\pi^2} - \frac{1}{x^2+16\pi^2} - \cdots\right)$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à co-secante hiperbólica do ângulo $x$. O valor de $\pi$ pode ser obtido usando os métodos `calculaPi` e `calculaPiDeOutraForma` (veja os exercícios 7.53 e 7.54) ou usando a constante `Math.PI`. Veja também o exercício 7.71.

**Exercício 7.73:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoTangenteHiperbólica` que calcule a série $x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \frac{x^9}{9} + \frac{x^{11}}{11} + \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método e $|x|$ deve ser menor que 1. O resultado da série, se calculada infinitamente, será igual ao arcotangente hiperbólico do ângulo $x$. O valor de $x^y$ pode ser calculado com o método `elevadoA` (exercício 7.30).

**Exercício 7.74:** ★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoCotangenteHiperbólica` que calcule a série $\frac{1}{x} + \frac{1}{3x^3} + \frac{1}{5x^5} + \frac{1}{7x^7} + \frac{1}{9x^9} + \frac{1}{11x^{11}} + \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método, e $|x|$ deve ser maior que 1. O resultado da série, se calculada infinitamente, será igual ao arcocotangente hiperbólico do ângulo $x$. O valor de $x^y$ pode ser calculado com o método `elevadoA` (exercício 7.30).

**Exercício 7.75:** ★★★

Escreva um programa em Java que demonstre que a série $\frac{1}{1^2 \times 3^2} + \frac{1}{3^2 \times 5^2} + \frac{1}{5^2 \times 7^2} + \frac{1}{7^2 \times 9^2} + \frac{1}{9^2 \times 11^2} + \cdots$ é aproximadamente igual a $\frac{\pi^2 - 8}{16}$. Use para isto o método `calculaPi` (exercício 7.53). Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores.

**Exercício 7.76:** ★★★

Escreva um programa em Java que demonstre que a série $\frac{1}{1^2 \times 2^2 \times 3^2} + \frac{1}{2^2 \times 3^2 \times 4^2} + \frac{1}{3^2 \times 4^2 \times 5^2} + \frac{1}{4^2 \times 5^2 \times 6^2} + \cdots$ é aproximadamente igual a $\frac{4\pi^2 - 39}{16}$. Use para isto o método `calculaPi` (exercício 7.53). Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores.

**Exercício 7.77:** ★★★

Escreva um programa em Java que demonstre que a série $\frac{1}{1^3} + \frac{1}{3^3} - \frac{1}{5^3} - \frac{1}{7^3} + \frac{1}{9^3} + \frac{1}{11^3} - \cdots$ é aproximadamente igual a $\frac{3\pi^3 - \sqrt{2}}{128}$. Use para isto o método `calculaPi` (exercício 7.53) e a constante `raizDe2` da classe `ConstantesMatematicas` (listagem 5.4). Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores. *Dica:* pode ser mais simples calcular vários termos da série em cada iteração do laço do que criar uma lógica que determine corretamente o sinal de cada termo.

**Exercício 7.78:** ★★★

Escreva um programa em Java que demonstre que a série $1 - \frac{1}{4} + \frac{1}{7} - \frac{1}{10} + \frac{1}{13} - \frac{1}{16} + \cdots$ é aproximadamente igual a $\frac{\pi\sqrt{3}}{9} + \frac{1}{3}ln(2)$. Use para isto o método `calculaPi` (exercício 7.53), a constante `raizDe3` da classe `ConstantesMatematicas` (listagem 5.4) e o resultado do exercício 7.40. Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores.

**Exercício 7.79:** ★★★

Escreva um programa em Java que demonstre que a série $x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \cdots$ é aproximadamente igual a $\ln(1+x)$ se $-1 < x \leq +1$. Use para isto os métodos `elevadoA` (exercício 7.30) e `logaritmoNatural` (exercício 7.85). Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores. Escreva o programa de forma a impedir que o usuário escolha valores de $x$ fora da faixa de valores válidos.

**Exercício 7.80:** ★★★

Escreva um programa em Java que demonstre que a série $1 - \frac{1}{5} + \frac{1}{9} - \frac{1}{13} + \frac{1}{17} - \frac{1}{21} + \cdots$ é aproximadamente igual a $\frac{\pi\sqrt{2}}{8} + \frac{\sqrt{2}\times\ln(1+\sqrt{2})}{4}$. Use para isto o método `calculaPi` (exercício 7.53), a constante `raizDe2` da classe `ConstantesMatematicas` (listagem 5.4) e o resultado do método `logaritmoNatural` (exercício 7.85). Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores.

**Exercício 7.81:** ★★★

Escreva um programa em Java que demonstre que a série $\frac{1}{2} - \frac{1}{5} + \frac{1}{8} - \frac{1}{11} + \frac{1}{14} - \frac{1}{17} + \cdots$ é aproximadamente igual a $\frac{\pi\sqrt{3}}{9} - \frac{1}{3}\ln(2)$. Use para isto o método `calculaPi` (exercício 7.53), a constante `raizDe3` da classe `ConstantesMatematicas` (listagem 5.4) e o resultado do exercício 7.40. Permita ao usuário entrar o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores.

**Exercício 7.82:** ★★★

Escreva uma versão recursiva do programa que calcula a seqüência que converge para $1/2$ (exercício 7.37).

**Exercício 7.83:** ★★★

Escreva uma versão recursiva do programa que calcula a seqüência que converge para $3/4$ (exercício 7.38).

**Exercício 7.84:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoSeno` que calcule a série $x + \frac{1}{2}\frac{x^3}{3} + \frac{1\times3}{2\times4}\frac{x^5}{5} + \frac{1\times3\times5}{2\times4\times6}\frac{x^7}{7} + \frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{x^9}{9} + \cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao arco seno do ângulo $x$ em radianos, para $|x| < 1$. O valor de $x^y$ pode ser calculado com o método `elevadoA` (exercício 7.30). *Dica:* escreva dois métodos auxiliares `multiplicaSériePar` e `multiplicaSérieImpar` que possam calcular $2\times4\times6\times8...N$ e $1\times3\times5\times7...N$.

**Exercício 7.85:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `logaritmoNatural` que calcule a série $2\times\left[\left(\frac{x-1}{x+1}\right) + \frac{1}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{1}{5}\left(\frac{x-1}{x+1}\right)^5 + \frac{1}{7}\left(\frac{x-1}{x+1}\right)^7 + \cdots\right]$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao logaritmo natural de $x$, para $x > 0$. Veja também o exercício 7.30.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.86:** ★★★★

Escreva um programa em Java que demonstre que a série $x-(1+\frac{1}{2})\times x^2+(1+\frac{1}{2}+\frac{1}{3})\times x^3-(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4})\times x^4+(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5})\times x^5-\cdots$ é aproximadamente igual a $\ln(1+x)/(1+x)$, para $|x|<1$. Use para isto os métodos `elevadoA` (exercício 7.30) e `logaritmoNatural` (exercício 7.85). Permita ao usuário entrar o valor $x$ e o número $N$ de termos da série pelo teclado, para poder experimentar com vários valores. Escreva o programa de forma a impedir que o usuário escolha valores de $x$ fora da faixa de valores válidos.

**Exercício 7.87:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoCoseno` que calcule a série $\frac{\pi}{2}-\left[x+\frac{1}{2}\frac{x^3}{3}+\frac{1\times3}{2\times4}\frac{x^5}{5}+\frac{1\times3\times5}{2\times4\times6}\frac{x^7}{7}+\frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{x^9}{9}+\cdots\right]$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao arco co-seno do ângulo $x$ em radianos, para $|x|<1$. O valor de $x^y$ pode ser calculado com o método `elevadoA` (exercício 7.30). *Dica:* parte da equação mostrada acima pode ser calculada com o método `arcoSeno` (exercício 7.84).

**Exercício 7.88:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoCosecante` que calcule a série $\frac{1}{x}+\frac{1}{2}\frac{1}{x^3}+\frac{1\times3}{2\times4}\frac{1}{x^5}+\frac{1\times3\times5}{2\times4\times6}\frac{1}{x^7}+\frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{1}{x^9}+\cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao arco co-secante do ângulo $x$ em radianos, para $|x|>1$. O valor de $x^y$ pode ser calculado com o método `elevadoA` (exercício 7.30). *Dica:* veja os métodos que devem ter sido criados como parte da resolução do exercício 7.84.

**Exercício 7.89:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoSecante` que calcule a série $\frac{\pi}{2}-\left[\frac{1}{x}+\frac{1}{2}\frac{1}{x^3}+\frac{1\times3}{2\times4}\frac{1}{x^5}+\frac{1\times3\times5}{2\times4\times6}\frac{1}{x^7}+\frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{1}{x^9}+\cdots\right]$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual à arco secante do ângulo $x$ em radianos, para $|x|>1$. O valor de $x^y$ pode ser calculado com o método `elevadoA` (exercício 7.30). *Dica:* parte da equação mostrada acima pode ser calculada com o método `arcoCosecante` (exercício 7.88).

**Exercício 7.90:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `logaritmoNaturalDeOutraForma` que calcule a série $(\frac{x-1}{x})+\frac{1}{2}(\frac{x-1}{x})^2+\frac{1}{3}(\frac{x-1}{x})^3+\frac{1}{4}(\frac{x-1}{x})^4+\cdots$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao logaritmo natural de $x$, para $x>\frac{1}{2}$. Escreva o método `logaritmoNaturalDeOutraForma` de forma que se um valor $x$ maior que zero e menor que $\frac{1}{2}$ for passado, o método chame o método `logaritmoNatural` (listagem 7.85) para calcular o logaritmo. Veja também o exercício 7.30.

**Exercício 7.91:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `secanteQuadrada` que calcule a série $4\times\left[(\frac{1}{(\pi-2x)^2})+(\frac{1}{(\pi+2x)^2})+(\frac{1}{(3\pi-2x)^2})+(\frac{1}{(3\pi+2x)^2})+(\frac{1}{(5\pi-2x)^2})+(\frac{1}{(5\pi+2x)^2})+\cdots\right]$ com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. O resultado da série, se calculada infinitamente, será igual ao quadrado da secante de $x$.

**Exercício 7.92:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `calculaPiComMétodoDeEuler` que calcule a série $2\times\left(1+\frac{1}{1\times3}+\frac{1\times2}{1\times3\times5}+\frac{1\times2\times3}{1\times3\times5\times7}+\frac{1\times2\times3\times4}{1\times3\times5\times7\times9}+\frac{1\times2\times3\times4\times5}{1\times3\times5\times7\times9\times11}+\cdots\right)$ com $N$ termos, sendo que o valor de $N$ deve ser fornecido como argumento ao método. O resultado da série, se calculada infinitamente, será igual a $\pi$. Veja também os exercícios 7.53 e 7.54. *Dica:* para valores muito grandes de $N$ é possível que o denominador seja igual a $Infinity$, fazendo com que a somatória passe a valer $NaN$. Tente encontrar o maior valor de $N$ que dá um resultado diferente de $NaN$.

**Exercício 7.93:** ★★★★

O valor $\frac{2}{\pi}$ pode ser calculado com a seguinte série infinita:

$$\frac{2}{\pi}=\frac{\sqrt{2}}{2}\,\frac{\sqrt{2+\sqrt{2}}}{2}\,\frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2}\,\frac{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}}{2}\,\frac{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}}}{2}\cdots$$

Escreva um programa em Java que calcule o valor $\frac{2}{\pi}$ usando a série acima, com $N$ termos, sendo que o valor de $N$ pode ser lido do teclado. *Dica:* um valor temporário calculado para um termo da série pode ser reutilizado para o cálculo do próximo valor.

**Exercício 7.94:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoTangente` que calcule uma das três séries

$$\begin{cases} -\frac{\pi}{2}-\frac{1}{x}+\frac{1}{3x^3}-\frac{1}{5x^5}+\frac{1}{7x^7}-\cdots & \text{se } x\le 1 \\ x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\frac{x^9}{9}-\cdots & \text{se } -1<x<1 \\ +\frac{\pi}{2}-\frac{1}{x}+\frac{1}{3x^3}-\frac{1}{5x^5}+\frac{1}{7x^7}-\cdots & \text{se } x\ge 1 \end{cases}$$

com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. A série a ser calculada depende do valor de $x$. O resultado da série, se calculada infinitamente, será igual ao arcotangente do ângulo $x$ em radianos. *Dicas:* veja que a diferença entre duas das séries é somente um sinal, e tente fazer subrotinas estáticas privadas que sejam usadas por este método.

**Exercício 7.95:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoCotangente` que calcule uma das três séries

$$\begin{cases} \pi+\frac{1}{x}-\frac{1}{3x^3}+\frac{1}{5x^5}-\frac{1}{7x^7}+\cdots & \text{se } x\le 1 \\ \frac{\pi}{2}-\left(x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\frac{x^9}{9}-\cdots\right) & \text{se } -1<x<1 \\ \frac{1}{x}-\frac{1}{3x^3}+\frac{1}{5x^5}-\frac{1}{7x^7}+\cdots & \text{se } x\ge 1 \end{cases}$$

com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. A série a ser calculada depende do valor de $x$. O resultado da série, se calculada infinitamente, será igual ao arcocotangente do ângulo $x$ em radianos. *Dicas:* veja que a diferença entre duas das séries é somente um sinal, e tente fazer subrotinas estáticas privadas que sejam usadas por este método.

**Exercício 7.96:** ★★★★

Escreva na classe `SeriesMatematicas` o método estático `arcoSenoHiperbólico` que calcule uma das três séries

$$\begin{cases} -[\ln|2x| + (\frac{1}{2}\frac{1}{2x^2} - \frac{1\times3}{2\times4}\frac{1}{4x^4} + \frac{1\times3\times5}{2\times4\times6}\frac{1}{6x^6} - \frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{1}{8x^8} + \cdots)] & \text{se } x \leq 1 \\ x - \frac{1}{2}\frac{x^3}{3} + \frac{1\times3}{2\times4}\frac{x^5}{5} - \frac{1\times3\times5}{2\times4\times6}\frac{x^7}{7} + \frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{x^9}{9} - \cdots & \text{se } -1 < x < 1 \\ +[\ln|2x| + (\frac{1}{2}\frac{1}{2x^2} - \frac{1\times3}{2\times4}\frac{1}{4x^4} + \frac{1\times3\times5}{2\times4\times6}\frac{1}{6x^6} - \frac{1\times3\times5\times7}{2\times4\times6\times8}\frac{1}{8x^8} + \cdots)] & \text{se } x \geq 1 \end{cases}$$

com $N$ termos, sendo que os valores de $N$ e $x$ devem ser fornecidos como argumentos ao método. A série a ser calculada depende do valor de $x$. O resultado da série, se calculada infinitamente, será igual ao arco seno hiperbólico do ângulo $x$. *Dicas:* veja que a diferença entre duas das séries é somente um sinal, e tente fazer subrotinas estáticas privadas que sejam usadas por este método.

**Exercício 7.97:** ★★★★

Escreva uma versão recursiva do programa que calcula a seqüência que converge para $ln(2)$ (exercício 7.40) de forma recursiva.

**Exercício 7.98:** ★★★★

Escreva uma versão recursiva do método que calcula $\pi$ de outra forma (exercício 7.54) na classe `SeriesMatematicas`.

**Exercício 7.99:** ★★★★

Escreva uma versão recursiva do método estático `senoDeXSobreX` (exercício 7.64) na classe `SeriesMatematicas`.

**Exercício 7.100:** ★★★★

Escreva uma versão recursiva do método estático `arcoCoseno` (exercício 7.87) na classe `SeriesMatematicas`.

**Exercício 7.101:** ★★★★

Escreva uma versão recursiva do método estático `arcoCosecante` (exercício 7.88) na classe `SeriesMatematicas`.

**Exercício 7.102:** ★★★★

Escreva uma versão recursiva do método estático `arcoSecante` (exercício 7.89) na classe `SeriesMatematicas`.

**Exercício 7.103:** ★★★★

Escreva uma versão otimizada do método `senoDeOutraForma` da classe `SeriesMatematicas` (veja o exercício 7.57). Escreva um programa que mostre que a versão otimizada é realmente mais rápida que a não-otimizada.

**Exercício 7.104:** ★★★★

Escreva uma versão otimizada do método `cosenoDeOutraForma` da classe `SeriesMatematicas` (veja o exercício 7.61). Escreva um programa que mostre que a versão otimizada é realmente mais rápida que a não-otimizada.

**Exercício 7.105:** ★★★★★

O valor de $1/\pi$ pode ser calculado por qualquer uma das séries ou seqüências abaixo:

$$\frac{1}{\pi} = \sum_{n=0}^{n=\infty} \frac{5+42n}{2^{12n+4}} \left( \frac{(2n)!}{(n!)^2} \right)^3$$

$$\frac{1}{\pi} = \frac{1}{72} \sum_{n=0}^{n=\infty} (-1^n) \frac{(4n)!}{(n!)^4 4^{4n}} \frac{23+260n}{18^{2n}}$$

$$\frac{1}{\pi} = \frac{1}{3528} \sum_{n=0}^{n=\infty} (-1^n) \frac{(4n)!}{(n!)^4 4^{4n}} \frac{1123+21460n}{882^{2n}}$$

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801} \sum_{n=0}^{n=\infty} \frac{(4n)!}{(n!)^4} \frac{1103+26390n}{396^{4n}}$$

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{n=0}^{n=\infty} \frac{(4n)!}{(n!)^4 4^{4n}} \frac{1103+26390n}{99^{4n}}$$

$$\frac{1}{\pi} = 12 \times \sum_{n=0}^{n=\infty} (-1^n) \frac{(6n)!}{(n!)^3 \times (3n)!} \frac{13591409+545140134n}{640320^{(3n+\frac{3}{2})}}$$

$$\frac{1}{\pi} = 12 \times \sum_{n=0}^{n=\infty} (-1^n) \frac{(6n)!}{(n!)^3 \times (3n)!} \frac{A+Bn}{C^{(n+\frac{1}{2})}}$$

onde

$$\begin{aligned} A &= 212175710912\sqrt{61} + 1657145277365 \\ B &= 13773980892672\sqrt{61} + 107578229802750 \\ C &= (5280 \times (236674 + 30303\sqrt{61}))^3 \end{aligned}$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $1/\pi$. Tente avaliar qual dos métodos é o mais eficiente: sabendo que $1/\pi$ é aproximadamente igual a 0.3183098861837907, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.106:** ★★★★★

O valor de $\pi$ também pode ser calculado indiretamente por qualquer uma das séries ou seqüências abaixo:

$$\frac{\pi^2}{18} = \sum_{n=1}^{n=\infty} \left( \frac{1}{n^2} \frac{(n!)^2}{(2n)!} \right)$$

$$10 - \pi^2 = \sum_{n=1}^{n=\infty} \left( \frac{1}{n^3(n+1)^3} \right)$$

$$\frac{\pi - 3}{6} = \sum_{n=1}^{n=\infty} \frac{(-1)^{n+1}}{36n^2 - 1}$$

$$\pi - 3 = \sum_{n=1}^{n=\infty} \frac{(-1)^{n+1}}{n(n+1)(2n+1)}$$

$$\frac{2\pi\sqrt{3}}{27} + \frac{4}{3} = \sum_{n=0}^{n=\infty} \frac{(n!)^2}{(2n)!}$$

$$\pi + 3 = \sum_{n=1}^{n=\infty} \frac{n(n!)^2 2^n}{(2n)!}$$

$$\frac{32 - 3\pi^3}{64} = \sum_{n=1}^{n=\infty} \frac{1}{(4n^2-1)^3}$$

$$\frac{\pi^4 + 30\pi^2 - 384}{768} = \sum_{n=1}^{n=\infty} \frac{1}{(4n^2-1)^4}$$

$$\pi\sqrt{2} = \sum_{n=0}^{n=\infty} \left( \frac{4}{6n+1} + \frac{1}{6n+3} + \frac{1}{6n+5} \right) \frac{(-1)^n}{8^n}$$

$$\frac{\pi^2}{18} = \sum_{n=0}^{n=\infty} \left( \frac{1}{(6n+1)^2} - \frac{3}{2(6n+2)^2} - \frac{1}{2(6n+3)^2} - \frac{3}{8(6n+4)^2} + \frac{1}{16(6n+5)^2} \right) \left( \frac{1}{64} \right)^n$$

$$\frac{8\pi^2}{9} = \sum_{n=0}^{n=\infty} \left( \frac{16}{(6n+1)^2} - \frac{24}{(6n+2)^2} - \frac{8}{(6n+3)^2} - \frac{6}{(6n+4)^2} + \frac{1}{(6n+5)^2} \right) \left( \frac{1}{64} \right)^n$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $\pi$. Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\pi$ é aproximadamente igual a 3.141592653589793, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca. Tente incluir no cálculo do tempo gasto o tempo necessário para converter o valor da série ou somatória para $\pi$.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 7.107:** ★★★★★

O logaritmo natural de 2 ($\ln(2)$) pode ser calculado por qualquer uma das séries ou seqüências abaixo:

$$\ln(2) = \sum_{n=1}^{n=\infty} \frac{1}{n2^n}$$

$$\ln(2) = \frac{1}{2}\sum_{n=1}^{n=\infty} \frac{1}{n(2n-1)}$$

$$\ln(2) = \frac{3}{4} - \frac{1}{4}\sum_{n=1}^{n=\infty} \frac{1}{n(n+1)(2n+1)}$$

$$\ln(2) = \frac{1}{2} + \sum_{n=0}^{n=\infty} \frac{1}{(2n+1)(2n+2)(2n+3)}$$

$$\ln(2) = \frac{1}{2} + \frac{1}{2}\sum_{n=1}^{n=\infty} \frac{1}{n(4n^2-1)}$$

$$\ln(2) = \frac{2}{3} + \frac{1}{3}\sum_{n=1}^{n=\infty} \frac{1}{n(16n^2-1)}$$

$$\ln(2) = \frac{3}{4} - \frac{1}{2}\sum_{n=1}^{n=\infty} \frac{1}{n(4n^2-1)^2}$$

$$\ln(2) = \frac{1}{2} + \sum_{n=1}^{n=\infty} \frac{(-1)^{n-1}}{n(4n^4+1)}$$

$$\ln(2) = \frac{2}{3}\sum_{n=0}^{n=\infty} \frac{1}{(2n+1)9^n}$$

$$\ln(2) = \frac{1327}{1920} + \frac{45}{4}\sum_{n=4}^{n=\infty} \frac{(-1)^n}{n(n^2-1)(n^2-4)(n^2-9)}$$

$$\ln(2) = \frac{2}{3} + \sum_{n=1}^{n=\infty} \left(\frac{1}{2n} + \frac{1}{2n+1} + \frac{1}{8n+4} + \frac{1}{16n+12}\right)\frac{1}{16^n}$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $\ln(2)$. Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\ln(2)$ é aproximadamente igual a 0.6931471805599453, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.108:** ★★★★★

O valor de $\pi$ também pode ser calculado por qualquer uma das séries ou seqüências abaixo:

$$\pi = \sum_{n=0}^{n=\infty} \left( \frac{2}{4n+1} + \frac{2}{4n+2} + \frac{1}{4n+3} \right) \frac{(-1)^n}{4^n}$$

$$\pi = \sum_{n=0}^{n=\infty} \left( \frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \frac{1}{16^n}$$

$$\pi = \sum_{n=0}^{n=\infty} \left( \frac{2}{8n+1} + \frac{2}{4n+2} + \frac{1}{4n+3} - \frac{1}{8n+10} - \frac{1}{8n+12} - \frac{1}{16n+28} \right) \frac{1}{16^n}$$

$$\pi = \frac{1}{64} \sum_{n=0}^{n=\infty} \left( -\frac{32}{4n+1} - \frac{1}{4n+3} + \frac{256}{10n+1} - \frac{64}{10n+3} - \frac{4}{10n+5} - \frac{4}{10n+7} + \frac{1}{10n+9} \right) \frac{(-1)^n}{1024^n}$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $\pi$. Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\pi$ é aproximadamente igual a 3.141592653589793, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.109:** ★★★★★

O valor de $\pi$ também pode ser calculado com qualquer uma das chamadas *fórmulas de Machin* mostradas abaixo:

$$\frac{\pi}{2} = 2 \arctan\left(\frac{1}{\sqrt{2}}\right) - \arctan\left(\frac{1}{\sqrt{8}}\right)$$

$$\frac{\pi}{4} = \arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{3}\right)$$

$$\frac{\pi}{4} = 2 \arctan\left(\frac{1}{2}\right) - \arctan\left(\frac{1}{7}\right)$$

$$\frac{\pi}{4} = 2 \arctan\left(\frac{1}{3}\right) + \arctan\left(\frac{1}{7}\right)$$

$$\frac{\pi}{4} = 4 \arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $\pi$, usando o método `arcoTangente` da classe `SeriesMatematicas` (exercício 7.94). Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\pi$ é aproximadamente igual a 3.141592653589793, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.110:** ★★★★★

A raiz quadrada de 2 e seu inverso podem ser calculados com qualquer uma das seguintes séries:

$$\sqrt{2} = 1 + \sum_{n=1}^{n=\infty} (-1)^{n-1} \frac{(2n-2)!}{n!(n-1)!2^{2n-1}}$$

$$\sqrt{2} = \prod_{n=1}^{n=\infty} \left(1 + \frac{(-1)^{n-1}}{2n-1}\right)$$

$$\frac{1}{\sqrt{2}} = 1 + \sum_{n=1}^{n=\infty} (-1)^{n} \frac{(2n-1)!}{n!(n-1)!2^{2n-1}}$$

$$\frac{1}{\sqrt{2}} = \prod_{n=1}^{n=\infty} \left(1 - \frac{1}{4(2n-1)^2}\right)$$

Onde $\prod$ indica que os termos da série devem ser multiplicados. Escreva um programa em Java que, usando as séries acima, calcule o valor de $\sqrt{2}$. Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\sqrt{2}$ é aproximadamente igual a 1.4142135623730950488, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.111:** ★★★★★

O valor do logaritmo de 2 ($\ln(2)$) também pode ser calculado com as variações das fórmulas de Machin mostradas abaixo:

$$\ln(2) = 2 \operatorname{arctanh}\left(\frac{1}{5}\right) + 2 \operatorname{arctanh}\left(\frac{1}{7}\right)$$

$$\ln(2) = 4 \operatorname{arctanh}\left(\frac{1}{5}\right) - 2 \operatorname{arctanh}\left(\frac{1}{17}\right)$$

$$\ln(2) = 4 \operatorname{arctanh}\left(\frac{1}{7}\right) + 2 \operatorname{arctanh}\left(\frac{1}{17}\right)$$

$$\ln(2) = 4 \operatorname{arctanh}\left(\frac{1}{6}\right) - 2 \operatorname{arctanh}\left(\frac{1}{99}\right)$$

$$\ln(2) = 18 \operatorname{arctanh}\left(\frac{1}{26}\right) - 2 \operatorname{arctanh}\left(\frac{1}{4801}\right) + 8 \operatorname{arctanh}\left(\frac{1}{8749}\right)$$

$$\ln(2) = 144 \operatorname{arctanh}\left(\frac{1}{251}\right) + 54 \operatorname{arctanh}\left(\frac{1}{449}\right) - 38 \operatorname{arctanh}\left(\frac{1}{4801}\right) + 62 \operatorname{arctanh}\left(\frac{1}{8749}\right)$$

$$\ln(2) = 72 \operatorname{arctanh}\left(\frac{1}{127}\right) + 54 \operatorname{arctanh}\left(\frac{1}{449}\right) + 34 \operatorname{arctanh}\left(\frac{1}{4801}\right) - 10 \operatorname{arctanh}\left(\frac{1}{8749}\right)$$

Escreva um programa em Java que, usando as séries acima, calcule o valor do logaritmo de 2, usando o método `arcoTangenteHiperbólica` da classe `SeriesMatematicas` (exercício 7.73). Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\ln(2)$ é aproximadamente igual a 0.6931471805599453, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.112:** ★★★★★

A constante de Euler ($\gamma$) pode ser calculada por qualquer uma das séries ou seqüências abaixo:

$$\gamma = \sum_{n=1}^{n=\infty} \left( \frac{1}{n} - ln\left(1+\frac{1}{n}\right)\right)$$

$$\begin{aligned}\gamma &= \frac{1}{2}\times\left(\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}+\cdots\right)-\frac{1}{3}\times\left(\frac{1}{1^3}+\frac{1}{2^3}+\frac{1}{3^3}+\frac{1}{4^3}+\cdots\right)+\\ &\frac{1}{4}\times\left(\frac{1}{1^4}+\frac{1}{2^4}+\frac{1}{3^4}+\frac{1}{4^4}+\cdots\right)-\frac{1}{5}\times\left(\frac{1}{1^5}+\frac{1}{2^5}+\frac{1}{3^5}+\frac{1}{4^5}+\cdots\right)+\cdots\end{aligned}$$

$$\begin{aligned}\gamma &= \frac{1}{2}\times\left(\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{4^2}+\frac{1}{5^2}+\cdots\right)+\frac{2}{3}\times\left(\frac{1}{2^3}+\frac{1}{3^3}+\frac{1}{4^3}+\frac{1}{5^3}+\cdots\right)+\\ &\frac{3}{4}\times\left(\frac{1}{2^4}+\frac{1}{3^4}+\frac{1}{4^4}+\frac{1}{5^4}+\cdots\right)+\frac{4}{5}\times\left(\frac{1}{2^5}+\frac{1}{3^5}+\frac{1}{4^5}+\frac{1}{5^5}+\cdots\right)+\cdots\end{aligned}$$

$$\begin{aligned}\gamma &= \frac{1}{2}-\frac{1}{3}+\\ &2\times\left(\frac{1}{4}-\frac{1}{5}+\frac{1}{6}-\frac{1}{7}\right)+\\ &3\times\left(\frac{1}{8}-\frac{1}{9}+\frac{1}{10}-\frac{1}{11}+\frac{1}{12}-\frac{1}{14}+\frac{1}{14}-\frac{1}{15}\right)+\\ &4\times\left(\frac{1}{16}-\frac{1}{17}+\frac{1}{18}-\frac{1}{19}+\frac{1}{20}-\frac{1}{21}+\frac{1}{22}-\cdots+\frac{1}{28}-\frac{1}{29}+\frac{1}{30}-\frac{1}{31}\right)+\cdots\end{aligned}$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $\gamma$. Tente avaliar qual dos métodos é o mais eficiente: sabendo que $\gamma$ é aproximadamente igual a 0.5772156649, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

**Exercício 7.113:** ★★★★★

A base dos logaritmos naturais ($e$) pode ser calculada por qualquer uma das séries ou seqüências abaixo:

$$e = \frac{2}{1}\left(\frac{4}{3}\right)^{\frac{1}{2}}\left(\frac{6\times8}{5\times7}\right)^{\frac{1}{4}}\left(\frac{10\times12\times14\times16}{9\times11\times13\times15}\right)^{\frac{1}{8}}\left(\frac{18\times20\times22\times24\times26\times28\times30\times32}{17\times19\times21\times23\times25\times27\times29\times31}\right)^{\frac{1}{16}}\cdots$$

$$e = \left(\frac{2}{1}\right)^{\frac{1}{2}}\left(\frac{2\times4}{3\times3}\right)^{\frac{1}{4}}\left(\frac{4\times6\times6\times8}{5\times5\times7\times7}\right)^{\frac{1}{8}}\left(\frac{8\times10\times10\times12\times12\times14\times14\times16}{9\times9\times11\times11\times13\times13\times15\times15}\right)^{\frac{1}{16}}\cdots$$

Escreva um programa em Java que, usando as séries acima, calcule o valor de $e$ (base dos logaritmos naturais). Tente avaliar qual dos métodos é o mais eficiente: sabendo que $e$ é aproximadamente igual a 2.718281828459045, execute os métodos até que a diferença entre o valor calculado e o valor constante seja muito pequena, e meça o tempo que cada um dos métodos demorou para atingir esta marca.

## 7.6 Exercícios complementares do capítulo 7

**Exercício 7.114:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeUnidadesDeComprimento` (listagem 5.6), imprima as distâncias em milhas e pés correspondentes às distâncias em metros de zero a cem metros. Veja também o exercício 5.22.

**Exercício 7.115:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeUnidadesDeArea` (exercício 5.7), imprima a área em acres e centímetros quadrados correspondentes aos valores em metros quadrados, de zero a dez mil metros quadrados, de cinquenta em cinquenta metros quadrados. Veja também o exercício 5.23.

**Exercício 7.116:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeUnidadesDeVolume` (exercício 5.8), imprima o volume em pés cúbicos e em centímetros cúbicos correspondentes aos valores em litros, de zero a mil litros, de vinte em vinte litros. Veja também o exercício 5.24.

**Exercício 7.117:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeUnidadesDeVelocidade` (exercício 5.9), imprima a velocidade em milhas por horas e pés por segundo correspondentes às velocidades em quilômetros por hora, de zero a cinquenta quilômetros por hora, de meio em meio quilômetro por hora. Veja também o exercício 5.25.

**Exercício 7.118:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeUnidadesDeTempo` (exercício 5.10), imprima o tempo em segundos, horas, semanas e anos correspondentes ao tempo em dias, de um a trinta e um dias. Veja também o exercício 5.26.

**Exercício 7.119:** ⋆

Escreva um programa em Java que, usando a classe `ConversaoDeTemperatura` (exercício 5.11), imprima as temperatura em graus Kelvin e Fahrenheit correspondentes aos graus Celsius entre zero e cem graus. Veja também o exercício 5.27.

**Exercício 7.120:** ⋆

Escreva uma aplicação em Java que crie e use instâncias das classes `EscolhaComWhile` (listagem 7.2) ou `EscolhaComDoWhile` (listagem 7.3).

**Exercício 7.121:** ⋆

Modifique as classes `JogoDeAdivinhacao` e `DemoJogoDeAdivinhacao` (listagens 7.4 e 7.5) para que o número de tentativas seja também variável, isto é, que não seja fixo na classe `JogoDeAdivinhacao`.

**Exercício 7.122:** ⋆

O resultado da execução da classe `SomatoriaRecursiva` (listagem 7.9 mostra erros de concordância quando o valor de $N$ é igual a 1 ou zero - estes erros ocorreram porque o método `somatória` tinha que ser mantido simples para as explicações necessárias. Corrija o método `somatória` para que as mensagens sejam impressas corretamente.

**Exercício 7.123:** ⋆

Explique por que o método `somatória` na classe `SomatoriaRecursiva` (listagem 7.9) retorna um valor do tipo `long` ao invés de `int`.

**Exercício 7.124:** ⋆⋆

Um problema potencial das classes `EscolhaComWhile` (listagem 7.2) e `EscolhaComDoWhile` (listagem 7.2) é que seus construtores não verificam se o valor inicial é menor do que o final - se uma instância for construída com os valores `(1000,-1000)`, o método `escolhe` nunca terminará !
Reescreva os construtores de forma que se o valor inicial da faixa for maior do que o final, o construtor trocará os valores, fazendo com que o final vire o inicial e vice-versa.

**Exercício 7.125:** ⋆⋆

Modifique o laço `while` entre as linhas 45 e 49 da classe `DemoWhile` (listagem 7.1) para que o laço calcule e imprima os fatores de dois até o valor 9000000000000000000L (nove seguido de dezoito zeros, do tipo `long`). O que acontecerá ? Explique.

**Exercício 7.126:** ⋆⋆

Reescreva a classe `EscolhaComWhile` (listagem 7.2) para que o método `escolha` seja estático, recebendo os valores inicial e final da faixa de valores que podem ser escolhidos como argumentos. O que deverá ser feito do construtor da classe ?

**Exercício 7.127:** ⋆⋆

Escreva um programa em Java que verifique a diferença de tempo de execução de um contador de um a um milhão usando laços `while`, `do-while` e `for`. Existem diferenças ? Use também um laço infinito (cuja condição de avaliação seja `true`) que seja terminado por um `if` associado a um `break` (veja listagem 7.6, linhas 41 a 46) para as comparações. Explique as diferenças, se estas existirem e forem significantes.

**Exercício 7.128:** ⋆⋆

Escreva um programa em Java que verifique a diferença de tempo de execução do cálculo do trigésimo elemento da série de Fibonacci, usando o algoritmo recursivo (listagem 7.10) e o algoritmo não-recursivo (exercício 7.21). **Evite tentar calcular o tempo de execução do algoritmo recursivo de Fibonacci para valores muito grandes.**

# Capítulo 8

# Reutilização de Classes

## 8.1 Introdução

*reutilização de classes*

Uma das características mais interessantes de linguagens de programação orientadas a objetos é a capacidade de facilitar a reutilização de código - o aproveitamento de classes e seus métodos que já estejam escritos e que já tenham o seu funcionamento testado e comprovado. Reutilização de código diminui a necessidade de escrever novos métodos e classes, economizando o trabalho do programador e diminuindo a possibilidade de erros.

Linguagens procedurais podem implementar reutilização de código com funções que podem ser chamadas a partir de vários programas diferentes. Java e outras linguagens de programação orientadas a objetos vão mais além, permitindo a criação de classes baseadas em outras. As classes criadas com esta técnica poderão conter os métodos das classes originais, além de poder adicionar comportamento específico da nova classe.

Como exemplo, consideremos um aluno de um curso universitário, que pode ser modelado por uma instância da classe `RegistroAcademico`, que contém campos para representar os dados do aluno e métodos para inicializar estes campos, imprimí-los, calcular a mensalidade devida, etc. Um aluno de pós-graduação não poderia ser modelado por uma instância da mesma classe, uma vez que para alunos de pós-graduação devemos também representar dados sobre o título da tese e nome do orientador, por exemplo. Não seria prático incluir campos para representar estes dados na classe `RegistroAcademico` para que a mesma pudesse ser usada para modelar alunos de graduação e de pós-graduação simultaneamente, pois para cada aluno da graduação que fosse modelado por uma instância da classe, dois campos não estariam sendo utilizados, o que é um desperdício. Um programador poderia se sentir tentado a criar uma classe que pudesse representar qualquer tipo de aluno, o que seria desnecessariamente complicado e possivelmente de difícil manutenção.

A solução dada por linguagens de programação orientadas a objeto é criar uma classe nova, por exemplo, `RegistroAcademicoDePosGraduacao`, que contém os campos e métodos da classe base `RegistroAcademico` e os campos e métodos adicionais que diferenciam o uso e comportamento das duas classes. O código não é copiado diretamente da classe base para a nova classe, o que facilita a manutenção - em caso de mudança em algum método da classe base a nova classe não precisaria ser modificada manualmente.

Existem dois mecanismos básicos de reuso de classes em Java: delegação (ou composição) e herança. Com delegação, usamos uma instância da classe base como campo na nova classe, e com herança criamos a classe nova como uma extensão direta da classe base. Este capítulo



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

apresenta estes dois mecanismos com exemplos e discussões.

## 8.2 Delegação ou Composição

*delegação*
*composição*

O primeiro mecanismo de reaproveitamento de classes em Java é conhecido como *delegação* ou *composição*. Podemos criar novas classes que estendem uma outra classe base se incluirmos uma instância da classe base como um dos campos da nova classe, que será então *composta* de campos específicos e de uma instância de uma classe base. Para que os métodos da classe base possam ser executados, escreveremos métodos correspondentes na classe nova que chamam os da classe base, desta forma *delegando* a execução dos métodos.

Um exemplo prático é mostrado a seguir. Consideremos as classes `Data`, que representa uma data, e a classe `Hora`, que representa uma hora (incluindo segundos). Ambas as classes contém métodos que inicializam e verificam a validade de seus campos e imprimem os valores de seus campos em um formato apropriado. Com estas classes, podemos criar a classe `DataHora`, que representa simultaneamente uma data e uma hora, sem que seja necessário reescrever os campos e métodos contidos nas classes `Data` e `Hora`. A classe `DataHora` é mostrada na listagem 8.1.

Listagem 8.1: A classe `DataHora`, que reusa as classes `Data` e `Hora` através de delegação.

```
1  /**
2   * A classe DataHora, que reusa as classes Data e Hora através de delegação.
3   * A data e hora são representadas por instâncias das respectivas classes que estão
4   * embutidas na classe DataHora, e toda a interação entre esta classe e as embutidas
5   * é feita através da chamada de métodos das classes embutidas. Esta classe demonstra
6   * o conceito de reutilização de classes através de delegação ou composição.
7   */
8  class DataHora // declaração da classe
9    {
10   /**
11    * Declaração dos campos da classe. Estes campos são declarados como privados
12    * para que não possam ser acessados de fora da classe.
13    */
14   private Data estaData; // uma instância da classe Data representa o dia, mês e ano
15   private Hora estaHora; // uma instância da classe Hora representa a hora, minuto
16                          // e segundo
17
18  /**
19   * O construtor para a classe DataHora, que recebe argumentos para inicializar
20   * todos os campos que esta classe indiretamente contém, e chama os construtores
21   * das classes Data e Hora para inicializar os campos das instâncias destas classes.
22   * @param hora a hora
23   * @param minuto o minuto
24   * @param segundo o segundo
25   * @param dia o dia
26   * @param mês o mês
27   * @param ano o ano
28   */
29   DataHora(byte hora,byte minuto,byte segundo,byte dia,byte mês,short ano)
30     {
31     estaData = new Data(dia,mês,ano);
32     estaHora = new Hora(hora,minuto,segundo);
33     }
34
35  /**
36   * O construtor para a classe DataHora, que recebe argumentos para inicializar
37   * os campos que representam uma data. O construtor também inicializará os campos
38   * que representam uma hora, considerando que todos valem zero (meia-noite).
39   * De novo, os construtores das classes embutidas nesta serão chamados.
40   * @param dia o dia
41   * @param mês o mês
42   * @param ano o ano
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
43    */
44    DataHora(byte dia,byte mês,short ano)
45      {
46      estaData = new Data(dia,mês,ano);
47      estaHora = new Hora((byte)0,(byte)0,(byte)0); // cast necessário
48      }
49
50    /**
51     * O método toString não recebe argumentos, e retorna uma String contendo os valores
52     * dos campos da classe formatados. Os valores são obtidos através da chamada
53     * implícita aos métodos toString das instâncias das classes embutidas.
54     * @return uma String com os valores dos campos formatados.
55     */
56    public String toString()
57      {
58      return estaData+" "+estaHora;
59      }
60
61    } // fim da classe DataHora
```

Algumas características interessantes da classe `DataHora` (listagem 8.1) são:

- Não existem campos na classe `DataHora` a não ser uma instância da classe `Data` e uma da classe `Hora` - todas as informações que devem ser representadas por uma classe `DataHora` estarão contidas nestas instâncias. Campos adicionais poderiam ser declarados, se fosse necessário.
- O construtor completo da classe `DataHora` (linha 29) recebe seis argumentos, correspondentes ao dia, mês, ano, hora, minuto e segundo que devem ser representados pela classe `DataHora`, e repassa estes argumentos para os construtores que inicializarão as instâncias das classes `Data` e `Hora`. O construtor da classe `DataHora` *delega* aos outros construtores a inicialização dos campos. O mesmo acontece com o construtor parcial (linha 44), exceto que este considera que a hora é, por *default*, meia-noite.
- O método `toString` (linha 56) também delega o seu funcionamento aos métodos `toString` das classes `Data` e `Hora`, que retornarão Strings que são concatenadas para criar o resultado da chamada do método `toString` da classe `DataHora`.

*vantages da reutilização de classes*

Uma das vantagens da reutilização de classes pode ser vista com o exemplo da classe `DataHora`: a nova classe é capaz de representar simultaneamente uma data e hora sem ser muito complexa. A complexidade (capacidade de verificar se a data ou hora estão corretas, etc.) é implementada pelos métodos das classes `Data` e `Hora`, que são simplesmente reutilizados.

Um outro exemplo de reaproveitamento de classes existentes usando o mecanismo de delegação pode ser visto na classe `RegistroAcademicoDeGraduacao`, mostrada na listagem 8.2.

Listagem 8.2: A classe `RegistroAcademicoDeGraduacao`, que reusa a classe `Data` através de delegação.

```
1  /**
2   * A classe RegistroAcademicoDeGraduacao, que encapsula os dados de um aluno
3   * regularmente matriculado em um curso de graduação.
4   */
5  class RegistroAcademicoDeGraduacao
6    {
7   /**
8    * Declaração dos campos da classe
9    */
10   private String nomeDoAluno; // O nome do aluno
11   private Data dataDeNascimento; // a data de nascimento
12   private int númeroDeMatrícula; // O número de matrícula
```

```
13    private byte códigoDoCurso; // O código do curso do aluno (1..4)
14    private double percentualDeCobrança; // O percentual a ser cobrado do aluno, 0 a 100%
15   /**
16    * Declaração de alguns campos constantes desta classe. Estes campos são declarados
17    * como public para serem acessíveis de fora da classe, como static pois serão
18    * independentes de instâncias, e como final para que não sejam modificáveis.
19    */
20    public static final byte ARQUITETURA = 1;
21    public static final byte CIÊNCIADACOMPUTAÇÃO = 2;
22    public static final byte ENGENHARIADACOMPUTAÇÃO = 3;
23    public static final byte ZOOTECNIA = 4;
24
25   /**
26    * O construtor completo, que recebe argumentos para inicializar os campos da classe.
27    * @param n o nome do aluno
28    * @param d a data de nascimento
29    * @param m o número de matrícula
30    * @param c o código do curso
31    * @param p o percentual de bolsa
32    */
33    RegistroAcademicoDeGraduacao(String n,Data d,int m,byte c,double p)
34      {
35      nomeDoAluno = n;
36      dataDeNascimento = d;
37      númeroDeMatrícula = m;
38      códigoDoCurso = c;
39      percentualDeCobrança = p;
40      } // fim do construtor
41
42   /**
43    * O construtor que recebe argumentos para inicializar os campos da classe, menos o
44    * percentual de cobrança, que será considerado como 100%.
45    * @param n o nome do aluno
46    * @param m o número de matrícula
47    * @param c o código do curso
48    */
49    RegistroAcademicoDeGraduacao(String n,Data d,int m,byte c)
50      {
51      this(n,d,m,c,100); // chama o construtor
52      } // fim do construtor
53
54   /**
55    * O método calculaMensalidade calcula e retorna a mensalidade do aluno usando
56    * o código do seu curso e o percentual de cobrança.
57    * @return o valor da mensalidade do aluno
58    */
59    public double calculaMensalidade()
60      {
61      double mensalidade = 0; // valor deve ser inicializado
62      // Primeiro, dependendo do curso do aluno, determina a mensalidade básica. Note
63      // que estamos usando as constantes da classe, definidas anteriormente.
64      if (códigoDoCurso == ARQUITETURA) // Arquitetura
65        mensalidade = 450.00;
66      if (códigoDoCurso == CIÊNCIADACOMPUTAÇÃO) // Ciência da Computação
67        mensalidade = 500.00;
68      if (códigoDoCurso == ENGENHARIADACOMPUTAÇÃO) // Engenharia da Computação
69        mensalidade = 550.00;
70      if (códigoDoCurso == ZOOTECNIA) // Zootecnia
71        mensalidade = 380.00;
72      // Agora calcula o desconto com o percentual de cobrança. Se o percentual de
73      // cobrança for zero, a mensalidade também o será.
74      if (percentualDeCobrança == 0) mensalidade = 0;
75      // Senão, calculamos com uma fórmula simples.
76      else mensalidade = mensalidade * 100.0 / percentualDeCobrança;
77      return mensalidade;
78      } // fim do método calculaMensalidade
79
80   /**
81    * O método toString retorna uma String contendo os campos desta classe com uma
```

```
82    * formatação simples.
83    * @return os campos da classe, formatados
84    */
85   public String toString()
86     {
87     String resultado = ""; // adicionaremos os valores dos campos
88     resultado += "Matrícula: "+númeroDeMatrícula+" Nome: "+nomeDoAluno+"\n";
89     resultado += "Data de Nascimento: "+dataDeNascimento+"\n"; // chamada implícita ao
90                                                               // método toString da
91                                                               // classe Data.
92     resultado += "Código do Curso: "+códigoDoCurso+"\n";
93     resultado += "Percentual de Cobrança: "+percentualDeCobrança+"\n";
94     return resultado; // retorna a String concatenada
95     }
96
97   } // fim da classe RegistroAcademicoDeGraduacao
```

Alguns pontos interessantes da classe `RegistroAcademicoDeGraduacao` (listagem 8.2) são:

- A classe `RegistroAcademicoDeGraduacao` contém campos para representação dos dados de um aluno regularmente matriculado em um curso de graduação. Um dos campos (`dataDeNascimento`) é uma instância da classe `Data`, que deve ser construída e manipulada usando métodos dela mesma[1].
- Os construtores da classe `RegistroAcademicoDeGraduacao` recebem, como um dos seus argumentos, uma instância da classe `Data`. Esta instância será usada para inicializar o campo `dataDeNascimento` - note que o comportamento da inicialização é igual para todos os campos, independente de serem tipos nativos ou instâncias de classes.
- O método `toString` da classe `RegistroAcademicoDeGraduacao` cria uma String com os valores dos campos da classe concatenados. Para obter os valores formatados do campo `dataDeNascimento`, o método `toString` da classe `Data` é chamado implicitamente. Este é mais um exemplo de delegação: a classe `RegistroAcademicoDeGraduacao` delega à classe `Data` a formatação de seus campos em uma String, usando esta String como resultado parcial.
- Alguns campos são declarados como `public static final`, para funcionarem como constantes que podem ser usadas dentro da própria classe ou em outras classes e aplicações que usem a classe `RegistroAcademicoDeGraduacao`. Estas constantes representam os códigos dos cursos, evitando que o programador da classe ou programador usuário da classe tenha que memorizar o código de cada curso.

A classe `DemoRegistroAcademicoDeGraduacao`, mostrada na listagem 8.3, demonstra o uso de instâncias da classe `RegistroAcademicoDeGraduacao`.

Listagem 8.3: A classe `DemoRegistroAcademicoDeGraduacao`, que demonstra a criação e o uso de instâncias da classe `RegistroAcademicoDeGraduacao`.

```
1  /**
2   * A classe DemoRegistroAcademicoDeGraduacao, que demonstra a criação e o uso de
3   * instâncias da classe RegistroAcademicoDeGraduacao.
4   */
5  class DemoRegistroAcademicoDeGraduacao
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * algumas instâncias da classe RegistroAcademicoDeGraduacao, e demonstra seu uso.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
```

[1] Um dos campos é uma instância da classe `String`, mas não são necessários cuidados especiais para instâncias desta classe.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
11    *        de comando, mas que neste caso serão ignorados.
12    */
13    public static void main(String args[])
14      {
15      // Criaremos a primeira instância. Como o construtor da classe
16      // RegistroAcademicoDeGraduacao precisa de uma instância da classe Data como
17      // argumento, criaremos também esta instância explicitamente.
18      Data nascimentoDoMillôr = new Data((byte)10,(byte)4,(short)1940);
19      RegistroAcademicoDeGraduacao millôr =
20        new RegistroAcademicoDeGraduacao("Millôr Fernandes",
21                                         nascimentoDoMillôr,
22                                         34990917,
23                                         RegistroAcademicoDeGraduacao.CIÊNCIADACOMPUTAÇÃO,
24                                         0);
25      // Criaremos outra instância, desta vez usando uma instância da classe Data
26      // criada especialmente para ser passada como argumento para o construtor da
27      // classe RegistroAcademicoDeGraduacao.
28      RegistroAcademicoDeGraduacao laerte =
29        new RegistroAcademicoDeGraduacao("Laerte",
30                                         new Data((byte)22,(byte)9,(short)1958),
31                                         32990702,
32                                         RegistroAcademicoDeGraduacao.ARQUITETURA,
33                                         100);
34      // Imprimimos as instâncias (chamando implicitamente o método toString da
35      // classe RegistroAcademicoDeGraduacao)
36      System.out.println(millôr);
37      System.out.println(laerte);
38      }
39
40    } // fim da classe DemoRegistroAcademicoDeGraduacao
```

### 8.2.1 Delegação e modificadores de acesso

*delegação e modificadores de acesso*

Vimos na seção 2.6 que campos e métodos de classes podem ter modificadores de acesso que definem quais destes campos e métodos poderão ser acessados a partir de outras classes. Classes que contém instâncias de outras classes sofrem as restrições impostas pela classe contida da forma esperada, mais restrições impostas pela própria classe. Por exemplo, suponhamos que a classe `Data` tem seus campos públicos, e uma instância da classe `Data` é usada via delegação na classe `Evento`, e esta instância é declarada como `private`. De dentro da classe `Evento`, todos os campos da classe `Data` podem ser acessados e modificados, mas nenhuma classe que use a classe `Evento` poderá acessar diretamente estes campos.

Os efeitos das modificações de acesso com delegação são exemplificados pelas classes `Pessoa0`, `Funcionario0` e `DemoFuncionario0`, mostradas a seguir.

Listagem 8.4: A classe `Pessoa0`, que encapsula os dados de identificação de uma pessoa.

```
1  /**
2   * A classe Pessoa0, que encapsula os dados de identificação de uma pessoa.
3   * O nome da classe tem um zero pois esta classe ainda deverá ser modificada - no
4   * caso, alguns dos campos da classe são públicos, para demonstração do que acontece
5   * com os modificadores de acesso quando a classe é reutilizada em outra.
6   */
7  class Pessoa0 // declaração da classe
8    {
9   /**
10   * Declaração dos campos da classe
11   */
12   public String nome;     // o nome da pessoa - declarado público propositadamente
13   private int identidade; // o número da identidade da pessoa
14   private Data nascimento; // a data de nascimento da pessoa
15
16  /**
```

```
17    * O construtor da classe Pessoa0, que recebe argumentos para inicializar seus campos.
18    * @param n o nome da pessoa
19    * @param i o número da identidade da pessoa
20    * @param d a data de nascimento da pessoa
21    */
22   Pessoa0(String n,int i,Data d)
23     {
24     nome = n; identidade = i; nascimento = d;
25     }
26
27  /**
28   * O método toString não recebe argumentos, e retorna uma String contendo os valores
29   * dos campos da classe formatados.
30   * @return uma String com os valores dos campos formatados.
31   */
32   public String toString()
33     {
34     return "Nome: "+nome+"\nIdentidade:"+identidade+
35            "Data de Nascimento:"+nascimento;
36     }
37
38   } // fim da classe Pessoa0
```

A classe `Pessoa0` na listagem 8.4 encapsula os dados básicos de identificação de uma pessoa. O campo `nome` foi declarado como `public` propositalmente, para que possa ser acessado diretamente por classes que usem instância desta classe.

Listagem 8.5: A classe `Funcionario0`, que encapsula os dados básicos de um funcionário de uma empresa.

```
1  /**
2   * A classe Funcionario0, que encapsula os dados básicos de um funcionário de uma
3   * empresa. Esta classe em especial mostra o que acontece quando uma instância de
4   * outra classe (no caso, da classe Data), que tem modificadores de acesso, é usada
5   * com outros modificadores de acesso.
6   * O nome da classe tem um zero pois esta classe ainda será modificada.
7   */
8  class Funcionario0 // declaração da classe
9    {
10  /**
11   * Declaração dos campos da classe
12   */
13   private Pessoa0 funcionário; // identificação do funcionário, dada por uma instância
14                                // da classe Pessoa0
15   public Data admissão;  // data de admissão, declarada propositalmente como pública
16   private float salário; // salário do funcionário
17
18  /**
19   * O construtor para a classe Funcionario0, que recebe argumentos para inicializar
20   * todos os campos da classe.
21   * @param f uma instância da classe Pessoa0 com a identificação do funcionário
22   * @param a uma instância da classe Data com a data de admissão
23   * @param s um valor de ponto flutuante com o salário do funcionário
24   */
25   Funcionario0(Pessoa0 f,Data a,float s)
26     {
27     funcionário = f;
28     admissão = a;
29     salário = s;
30     }
31
32  /**
33   * O método toString não recebe argumentos, e retorna uma String contendo os valores
34   * dos campos da classe formatados.
35   * @return uma String com os valores dos campos formatados.
36   */
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
37   public String toString()
38     {
39     String resultado;
40     resultado = funcionário+"\n"; // chamada implícita ao método toString
41                                   // da classe Pessoa
42     resultado = resultado + "Data de admissão:"+admissão+"\n";
43     resultado = resultado + "Salário:"+salário+"\n";
44     return resultado;
45     }
46
47   } // fim da classe Funcionario0
```

A classe `Funcionario0` na listagem 8.5 encapsula os dados básicos sobre um funcionário de uma empresa. Esta classe usa uma instância da classe `Pessoa0` e uma da classe `Data` (via delegação), além de um valor do tipo `float` para representar o salário, que correspondem aos dados do funcionário. O campo `admissão` foi declarado como `public` propositalmente, para que possa ser acessado diretamente por classes que usem instância desta classe.

A classe `Data` não foi mostrada neste capítulo, mas assume-se que todos os seus campos são declarados como sendo `private`, podendo ser acessados somente através do construtor (para inicialização) e do método `toString` (para leitura).

Listagem 8.6: A classe `DemoFuncionario0`, que demonstra o uso de uma instância da classe `Funcionario0`.

```
1  /**
2   * A classe DemoFuncionario0, que demonstra o uso de uma instância da classe
3   * Funcionario0. Esta classe demonstra o que ocorre quando uma instância de uma
4   * classe cujos campos tem modificadores de acesso é reaproveitada (via delegação ou
5   * composição) dentro de outra classe que redeclara os modificadores de acesso.
6   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
7   */
8  class DemoFuncionario0 // declaração da classe
9    {
10   /**
11    * O método main permite a execução desta classe. Este método declara uma instância
12    * da classe Funcionario0 e tenta acessar alguns de seus campos e dos campos das
13    * classes que são usadas pela classe Funcionario0.
14    * @param argumentos os argumentos que podem ser passados para o método via linha
15    *        de comando, mas que neste caso serão ignorados.
16    */
17   public static void main(String[] argumentos)
18     {
19     // Criamos uma instância da classe Pessoa0, cujo construtor espera como terceiro
20     // argumento uma instância da classe Data
21     Pessoa0 pessoaChan = new Pessoa0("Patrick Chan",90235422,
22                                      new Data((byte)22,(byte)2,(short)1964));
23     // Criamos uma instância da classe Funcionario0, cujo construtor espera como
24     // primeiro argumento uma instância da classe Pessoa0, como segundo argumento uma
25     // instância da classe Data e como terceiro argumento um valor do tipo float.
26     Funcionario0 funcionárioChan = new Funcionario0(pessoaChan,
27                                                     new Data((byte)14,(byte)2,(byte)1990),
28                                                    (float)2400.00);
29     // Tentamos modificar diretamente o campo funcionário da instância. Teremos um
30     // erro de compilação pois o campo funcionário é privado da classe Funcionario0.
31     funcionárioChan.funcionário = new Pessoa0("José Ribamar",87124324,
32                                               new Data((byte)9,(byte)1,(short)1931));
33     // Tentamos modificar diretamente o nome do campo funcionário da instância. Teremos
34     // outro erro de compilação pois mesmo que o campo nome da classe Pessoa0 seja
35     // público, o campo funcionário da classe Funcionario0 é privado, impedindo o
36     // acesso.
37     funcionárioChan.funcionário.nome = "José Ribamar";
38     // Tentamos modificar o campo salário da instância. Teremos novamente um erro de
39     // compilação pois o campo salário é privado da classe Funcionario0.
```

```
40      funcionárioChan.salário = (float)1200.00;
41      // Tentamos modificar o campo admissão da instância. Desta vez não teremos erros
42      // pois o campo admissão é público da classe Funcionario0, podendo ser acessado
43      // diretamente usando o operador ponto.
44      funcionárioChan.admissão = new Data((byte)14,(byte)2,(byte)1989);
45      // Tentamos modificar diretamente o campo dia do campo admissão da instância.
46      // Teremos novamente um erro de compilação pois, mesmo que a instância admissão
47      // da classe Data tenha sido declarada como pública na classe Funcionario0, o campo
48      // dia é declarado como private na classe Data.
49      funcionárioChan.admissão.dia = (byte)22;
50      } // fim do método main
51
52    } // fim da classe DemoFuncionario0
```

A classe `DemoFuncionario0` (listagem 8.6) demonstra a criação e uso de uma instância da classe `Funcionario0`, com várias tentativas de acessar os campos das classes envolvidas. As tentativas de acesso aos campos e os efeitos são mostrados com detalhes a seguir.

- Nas linhas 31 e 32 da listagem 8.6, tentamos modificar diretamente o campo `funcionário` da classe `Funcionario0`, que, como foi declarado como `private`, não pode ser acessado diretamente.
- Na linha 37 tentamos modificar diretamente o campo `nome` da instância `funcionário`, declarada na classe `Funcionario0`. O campo `nome` da classe `Pessoa0` foi declarado como `public`, mas como a instância `funcionário` foi declarada como sendo `private` na classe `Funcionario0`, o acesso se torna impossível.
  Note que os métodos da classe `Funcionario0` podem acessar a instância `funcionário` diretamente, inclusive seu campo `nome`. Os métodos da classe `Funcionario0` não podem acessar o campo `identidade` da instância `funcionário` diretamente, pois este campo foi declarado como sendo `private`.
- Na linha 40 tentamos modificar diretamente o salário da instância da classe `Funcionario0`. O campo que representa o salário foi declarado como sendo privado: somente os métodos da classe `Funcionario0` podem acessá-lo ou modificá-lo.
- Na linha 44 tentamos modificar diretamente o campo `admissão`, que representa a data de admissão do funcionário. O campo `admissão` foi declarado como sendo público, podendo ser acessado e modificado de fora da classe, desta forma, não teremos um erro de compilação.
- Na linha 49 tentamos modificar diretamente o campo `dia` do campo `admissão`, o que resulta em um erro de compilação. Mesmo que o campo `admissão` tenha sido declarado como sendo público na classe `Funcionario0`, o campo `dia` da classe `Data` foi declarado como sendo `private`, não podendo ser modificado diretamente mesmo dentro de métodos da classe `Funcionario0`.

Em resumo, podemos notar que quando declaramos campos públicos em classes e reutilizamos estas classes dentro de outras, estes campos podem deixar de ser públicos; mas quando tempos campos que são privados nas classes e reusamos estas classes, declarando suas instâncias como públicas, os campos não passam a ser públicos.

### 8.2.2 Delegação e construtores

*delegação e construtores*

As regras para uso de construtores com classes que contém instâncias de outras classes é bem simples: se os construtores das classes usadas através da delegação deverem obrigatoriamente ser chamados, eles podem ser chamados em qualquer ponto da classe que os contém. Por exemplo, se for obrigatória a execução do construtor de uma instância da classe `Data` em uma classe



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

`Evento`, este construtor pode ser chamado em qualquer método da classe `Evento`, mas idealmente deve ser chamado a partir do construtor da própria classe `Evento`.

Nos exemplos de reuso através de delegação mostrados, não foi necessário chamar os construtores de classes reutilizadas pois as instâncias das classes foram criadas nos próprios aplicativos. Por exemplo, as instâncias da classes `Data` e `Pessoa0` são inicializadas e passadas como argumentos ao construtor da classe `Funcionario0`, desta forma não existe necessidade de chamar os construtores das classes `Data` e `Pessoa0` de dentro do construtor da classe `Funcionario0`.

Vale a pena lembrar que existe sempre mais de uma maneira de se criar uma classe em Java: poderíamos criar o construtor da classe `Funcionario0` recebendo os dados relativos à classe `Pessoa0` separados, como uma String (para o nome), um inteiro (para o número de identidade) e uma instância da classe `Data` (para o nascimento), e adicionalmente, os diversos campos necessários para representar uma data (admissão) e o salário como argumentos, e na declaração do construtor da classe `Funcionario0` repassaríamos estes argumentos para os respectivos construtores, como foi feito para a classe `DataHora` (listagem 8.1).

## 8.3 Herança

O mecanismo de reaproveitamento por delegação ou composição permite o reuso de classes já existentes como instâncias de novas classes. As classes originais ficam assim *contidas* na nova classe.

Reuso de classes via o mecanismo de delegação é útil quando consideramos que a classe que reusa instâncias de outras é *composta* das outras classes. Um bom exemplo é o da classe `DataHora`, que é composta das classes `Data` e `Hora`. Outros exemplos seriam dados pelas classes que simplesmente utilizam uma ou mais instâncias da classe `Data` juntamente com outros dados.

Nem sempre o mecanismo de delegação é o mais natural para reutilização de classes já existentes, embora seja simples. Em especial, quando queremos usar uma classe para servir de base à criação de outra mais especializada, a relação de composição imposta pelo uso do mecanismo de delegação acaba por criar soluções pouco naturais.

Como exemplo consideremos as classes `Pessoa` e `Funcionario`: uma instância de `Pessoa` pode ser declarada dentro da classe `Funcionario` para representar os dados da pessoa/funcionário. Se criássemos uma classe `ChefeDeDepartamento` considerando que um chefe de departamento é um funcionário que é responsável por um departamento, poderíamos declarar uma instância de `Funcionario` dentro da classe `ChefeDeDepartamento` e acrescentar alguns campos que diferenciam `ChefeDeDepartamento` de `Funcionario`.

*relação contém*

A relação entre as classes `ChefeDeDepartamento`, `Funcionario` e `Pessoa` seria, então, de composição: pela declaração dos seus campos veríamos que um `ChefeDeDepartamento` contém um `Funcionario`, que por sua vez contém uma `Pessoa`. Apesar de ser possível descrever as classes assim, a solução é pouco natural: embora possamos dizer que a classe `DataHora` *contém* uma data e uma hora, declarar que a classe `ChefeDeDepartamento` *contém* um funcionário soa meio artificial - na verdade, um chefe de departamento *é um tipo de* funcionário, que deve ter campos adicionais para representar dados que são específicos de um chefe de departamento, e métodos para manipular estes campos.

*herança*

Java oferece outra maneira de reutilizar classes, através do mecanismo de *herança*, que permite

que criemos uma classe usando outra como base e descrevendo ou implementando as diferenças e adições da classe usada como base, reutilizando os campos e métodos não-privados da classe-base. O mecanismo de herança é o mais apropriado para criar relações *é-um-tipo-de* entre classes. Com o mecanismo de herança, podemos declarar a classe `Funcionario` como sendo um tipo de `Pessoa`, e a classe `Funcionario` *herdará* todos os campos e métodos da classe `Pessoa`, não sendo necessária a sua redeclaração. Evidentemente uma classe herdeira pode acrescentar campos e métodos à classe original. Um primeiro exemplo de herança é visto a seguir, com as classes `Pessoa`, `Funcionario` e `ChefeDeDepartamento`.

*relação é-um-tipo-de*

Listagem 8.7: A classe `Pessoa`, que encapsula os dados de identificação de uma pessoa.

```
1  /**
2   * A classe Pessoa, que encapsula os dados de identificação de uma pessoa. A única
3   * diferença entre esta classe e a classe Pessoa0 é que todos os campos são declarados
4   * como private ao invés de public.
5   */
6  class Pessoa // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10   */
11   private String nome;     // o nome da pessoa
12   private int identidade; // o número da identidade da pessoa
13   private Data nascimento; // a data de nascimento da pessoa
14
15  /**
16   * O construtor da classe Pessoa, que recebe argumentos para inicializar seus campos.
17   * @param n o nome da pessoa
18   * @param i o número da identidade da pessoa
19   * @param d a data de nascimento da pessoa
20   */
21   Pessoa(String n,int i,Data d)
22     {
23     nome = n; identidade = i; nascimento = d;
24     }
25
26  /**
27   * O método toString não recebe argumentos, e retorna uma String contendo os valores
28   * dos campos da classe formatados.
29   * @return uma String com os valores dos campos formatados.
30   */
31   public String toString()
32     {
33     return "Nome: "+nome+"\nIdentidade: "+identidade+" "+
34            "\nData de Nascimento: "+nascimento;
35     }
36
37  } // fim da classe Pessoa
```

A classe `Pessoa` (mostrada na listagem 8.7) contém os campos para representar uma pessoa (com finalidade de identificação). Todos os campos são privados, devendo ser acessados através dos métodos da classe. Esta classe será usada como base para a classe `Funcionario`, mostrada na listagem 8.8.

Listagem 8.8: A classe `Funcionario`, que encapsula os dados básicos de um funcionário de uma empresa e herda da classe `Pessoa`.

```
1  /**
2   * A classe Funcionario, que encapsula os dados básicos de um funcionário de uma
3   * empresa. Esta classe herda da classe Pessoa, criando a relação "Funcionario é um
4   * tipo de Pessoa".
5   */
```

```
6  class Funcionario extends Pessoa // declaração da classe (herdando de Pessoa)
7    {
8   /**
9    * Declaração dos campos adicionais da classe. Note que não é necessário declarar
10   * um campo como sendo uma instância da classe Pessoa, como foi feito com o
11   * exemplo do mecanismo de delegação: os campos e métodos da classe Pessoa foram
12   * herdados (mas os campos deverão ser acessados indiretamente pois são privados da
13   * classe Pessoa).
14   */
15   private Data admissão;
16   private float salário;
17
18  /**
19   * O construtor para a classe Funcionario, que recebe argumentos para inicializar
20   * todos os campos da classe. Este construtor recebe os mesmos dados que seriam
21   * necessários para criar uma instância da classe Pessoa mais os que diferenciam
22   * a classe Funcionario da Pessoa. Passaremos os dados que representam uma Pessoa
23   * para o construtor da classe ancestral, via a palavra-chave super.
24   * @param nome o nome da pessoa
25   * @param id o número da identidade da pessoa
26   * @param nasc uma instância da classe Data com a data de nascimento
27   * @param adm uma instância da classe Data com a data de admissão
28   * @param sal o salário do funcionário
29   */
30   Funcionario(String nome,int id,Data nasc,Data adm,float sal)
31     {
32     super(nome,id,nasc);
33     admissão = adm;
34     salário = sal;
35     }
36
37  /**
38   * O método toString não recebe argumentos, e retorna uma String contendo os valores
39   * dos campos da classe formatados. Como é necessário ter os campos da classe
40   * ancestral formatados para retornar o resultado deste método, chamamos o método
41   * toString da classe ancestral via a palavra-chave super.
42   * @return uma String com os valores dos campos formatados.
43   */
44   public String toString()
45     {
46     String resultado;
47     resultado = super.toString()+"\n"; // chama o método toString da classe Pessoa
48     resultado = resultado + "Data de admissão: "+admissão+"\n";
49     resultado = resultado + "Salário: "+salário;
50     return resultado;
51     }
52
53  /**
54   * O método qualSalário retorna o valor do salário do funcionário. Este método será
55   * usado por qualquer outro método que queira somente acessar (ler) o valor numérico
56   * do salário.
57   * Este método foi declarado como final, e não poderá ser sobreposto por métodos
58   * com a mesma assinatura em classes herdeiras.
59   * @return o valor do salário do funcionário
60   */
61   final public float qualSalário()
62     {
63     return salário;
64     }
65
66   } // fim da classe Funcionario
```

Alguns dos pontos interessantes da classe `Funcionario` (listagem 8.8) são:

`extends`

- A classe `Funcionario`, na sua declaração, contém a palavra-chave `extends`, seguida de um nome de classe, no caso, `Pessoa`. A declaração com `extends` significa que a classe `Funcionario` conterá **todos** os campos e métodos da classe `Pessoa` **mais** os declarados

dentro da própria classe `Funcionario`[2].
Curiosamente, como os campos da classe `Pessoa` foram declarados como sendo privados da classe `Pessoa`, eles somente poderão ser acessados por métodos da classe `Pessoa` (que tenham sido declarados como `public`). Pelo mecanismo de herança, estes métodos estarão disponíveis para uso na classe `Funcionario` - desta forma os métodos herdados da classe `Pessoa` para a classe `Funcionario` podem acessar os campos da classe `Pessoa` que não poderiam ser acessados diretamente por métodos escritos na classe `Funcionario`.

*subclasse*

*superclasse*

- A classe que foi criada usando o mecanismo de herança é chamada de *subclasse* ou *classe herdeira*, enquanto que a classe usada como base para criar a classe herdeira é conhecida como *superclasse* ou *classe ancestral*.

`super`

- O construtor da classe `Funcionario` recebe, como argumentos, todos os dados necessários para criar uma instância da classe `Pessoa` mais os dados que são específicos da classe `Funcionario`. A inicialização destes dados é feita em duas partes: a inicialização dos campos que são herdados da classe `Pessoa` não pode ser feita diretamente pois estes campos foram declarados como sendo privados da classe `Pessoa`, então delegamos a inicialização destes campos ao construtor da classe `Pessoa`.
  Como o construtor da classe `Pessoa` não pode ser chamado diretamente, usamos a palavra-chave `super` seguida dos argumentos que seriam passados para o construtor da classe `Pessoa`. A palavra-chave `super`, seus usos e regras, serão vistos na seção 8.3.1.
- No método `toString` da classe `Funcionario` vemos outro exemplo de reuso através da herança - para obter uma String formatada com os campos de identificação do funcionário (definidos na classe `Pessoa`), chamamos o método `toString` da classe ancestral ou superclasse, usando novamente a palavra-chave `super`, e concatenamos os campos da própria classe `Funcionario`[3].

*métodos* `final`

- A classe `Funcionario` tem um método `qualSalário` que retorna o salário do funcionário, e é um marco da diferença entre as classes `Funcionario` e `Pessoa` - consideramos que somente `Pessoa`s que são `Funcionario`s tem o dado `salário` e um método para acessar o campo correspondente a esse dado. Este método é declarado como `final`, fazendo com que classes herdeiras não possam sobrepô-lo. Uma discussão mais detalhada sobre o uso de métodos `final` para impedir a herança deles é feita na seção 8.3.2 neste capítulo.

*uso de delegação e herança simultaneamente*

- Uma classe pode reusar outras usando ao mesmo tempo o mecanismo de herança e o de delegação: a classe `Funcionario` herda da classe `Pessoa` e ao mesmo tempo usa uma instância da classe `Data`. É importante, ao criar uma classe com o mecanismo de herança, considerar de qual classe ela deve herdar - não faria muito sentido, por exemplo, declarar `Funcionario` herdando da classe `Data` e contendo uma instância da classe `Pessoa` - a classe `Funcionario` pode ser considerada um tipo de `Pessoa`, mas definitivamente não é um tipo de `Data`.

Outro exemplo de herança é dado pela classe `ChefeDeDepartamento`, que herda da classe `Funcionario` e é mostrada na listagem 8.9.

[2] Existe uma certa discordância entre autores sobre se campos privados são herdados ou não. A especificação da linguagem Java diz que não, mas não explica como estes campos podem ser acessados indiretamente em classes herdeiras. Se por herança entendermos que a classe terá a sua própria versão dos campos herdados, campos privados não são considerados como herdados. Se por herança entendermos que existirá o acesso aos campos, direta ou indiretamente, os campos privados de superclasses são considerados herdados para as subclasses. Para simplificar, consideremos que os campos privados são herdados, mas como só podem ser acessados e modificados pelas classes que os declararam diretamente, não podem ser acessados diretamente pela classe herdeira.

[3] Na verdade, o método `toString` das instâncias da classe `Data` são chamados implicitamente quando a String é concatenada.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem 8.9: A classe `ChefeDeDepartamento`, que encapsula os dados básicos de um chefe de um departamento de uma empresa e herda da classe `Funcionario`.

```
1  /**
2   * A classe ChefeDeDepartamento, que encapsula os dados básicos de um chefe de um
3   * departamento de uma empresa. Esta classe herda da classe Funcionario, criando a
4   * relação "ChefeDeDepartamento é um tipo de Funcionario".
5   */
6  class ChefeDeDepartamento extends Funcionario // declaração da classe (herdando de
7                                                // Funcionario)
8    {
9   /**
10   * Declaração dos campos adicionais da classe. Note que não é necessário declarar
11   * um campo como sendo instância da classe Funcionario. Somente os campos que
12   * diferenciam um ChefeDeDepartamento de um Funcionario são necessários.
13   */
14   private String departamento;
15   private Data promoçãoAChefe;
16
17  /**
18   * O construtor para a classe ChefeDeDepartamento, que recebe argumentos para
19   * inicializar todos os campos da classe. Este construtor recebe os mesmos dados
20   * que seriam necessários para criar uma instância da classe Pessoa mais os que
21   * diferenciam a classe Funcionario da classe Pessoa e os que diferenciam a classe
22   * ChefeDeDepartamento da classe Funcionario. Passaremos os dados que representam
23   * um Funcionario para o construtor da classe ancestral, via a palavra-chave super.
24   * @param nome o nome da pessoa
25   * @param id o número da identidade da pessoa
26   * @param nasc uma instância da classe Data com a data de nascimento
27   * @param adm uma instância da classe Data com a data de admissão
28   * @param sal o salário do funcionário/chefe de departamento
29   * @param dep o departamento que este chefe chefia
30   * @param prom uma instância da classe Data com a data de promoção ao cargo
31   */
32   ChefeDeDepartamento(String nome,int id,Data nasc,
33                       Data adm,float sal,
34                       String dep,Data prom)
35     {
36     super(nome,id,nasc,adm,sal); // chama o construtor da classe Funcionario
37     departamento = dep;
38     promoçãoAChefe = prom;
39     }
40
41  /**
42   * O método toString não recebe argumentos, e retorna uma String contendo os valores
43   * dos campos da classe formatados. Como é necessário ter os campos da classe
44   * ancestral formatados para retornar todos os campos desta classe formatados,
45   * chamamos o método toString da classe ancestral via a palavra-chave super.
46   * @return uma String com os valores dos campos formatados.
47   */
48   public String toString()
49     {
50     String resultado;
51     resultado = super.toString()+"\n"; // chama o método toString da classe Funcionario
52     resultado = resultado + "Departamento:"+departamento+"\n";
53     resultado = resultado + "Data de promoção ao cargo:"+promoçãoAChefe;
54     return resultado;
55     }
56
57  /**
58   * O método qualDepartamento retorna o departamento do chefe de departamento. Este
59   * método será usado por qualquer outro método que queira acessar (ler) somente o
60   * valor do campo departamento.
61   * @return o departamento do chefe de departamento
62   */
63   public String qualDepartamento()
64     {
65     return departamento;
66     }
```

```
67
68  } // fim da classe ChefeDeDepartamento
```

Alguns pontos de interesse da listagem 8.9 são:

*herança de classes herdeiras*

- Consideramos que a classe `ChefeDeDepartamento` é um tipo de `Funcionario` e declaramos a classe como herdando da classe `Funcionario`. É perfeitamente possível (e, em muitos casos, necessário) declarar classes que descendem de classes que já descendem de outras. A classe herdeira terá acesso (direta ou indiretamente) a todos os campos e métodos das classes ancestrais, exceto em situações que serão discutidas na seção 8.3.2.
- O construtor da classe `ChefeDeDepartamento` recebe como argumentos dados necessários para inicializar uma instância da classe `Funcionario`, além de dados para inicializar os campos que são específicos da classe. Novamente, os dados necessários para inicializar uma instância da classe `Funcionario` são passados como argumentos para o construtor da classe `Funcionario` através da palavra-chave `super`. Já sabemos que o construtor da classe `Funcionario` fará o mesmo, chamando o construtor da classe `Pessoa`, que é a sua ancestral.
- De forma similar, o método `toString` da classe `ChefeDeDepartamento` executa explicitamente o método `toString` da classe ancestral (`Funcionario`), concatenando seu resultado com o dos métodos `toString` das classes `String` e `Data`, que serão chamados implicitamente.
- A classe `ChefeDeDepartamento` tem um método `qualDepartamento` que retorna o departamento chefiado, que marca a diferença entre as classes `ChefeDeDepartamento` e `Funcionario`.

A figura 8.1 ilustra o processo de herança de classes que já herdam de outras, usando como exemplos as classes criadas neste capítulo.

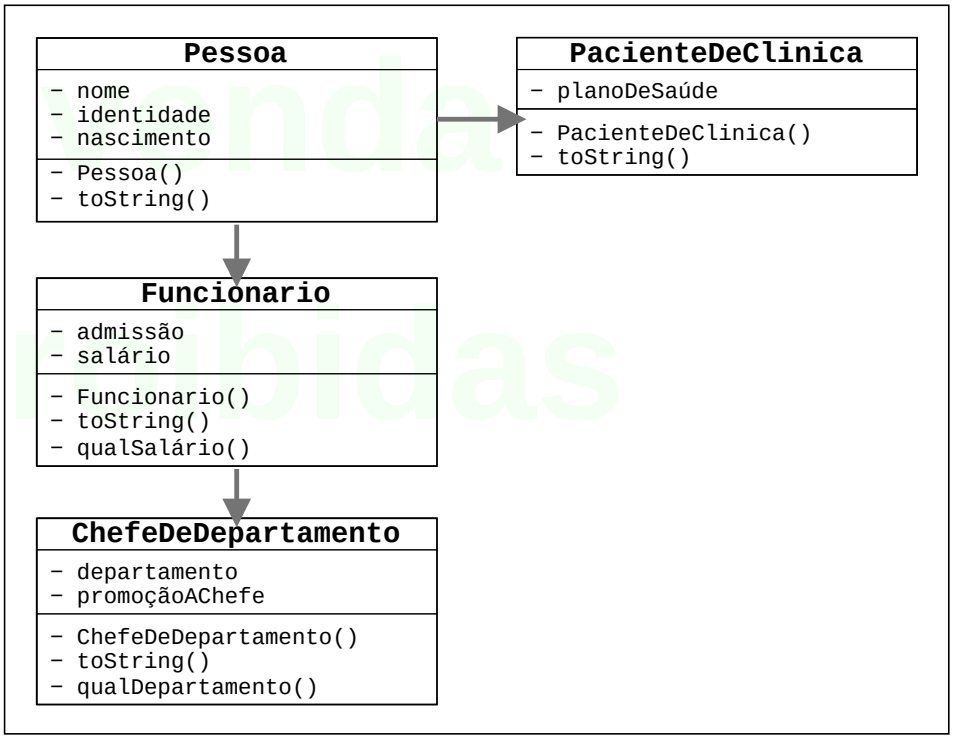



Figura 8.1: Esquema de herança envolvendo várias classes



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Alguns pontos de interesse da figura 8.1 são:

*detalhes de funcionamento do mecanismo de herança*

- A seta apontando de uma classe para outra denota que a classe que aponta herda da classe apontada. Desta forma, as classes `PacienteDeClinica` e `Funcionario` herdam da classe `Pessoa`. Alguns autores preferem desenhar a seta apontando da classe ancestral para a classe estendida.
- Uma subclasse conterá todos os campos e métodos declarados na superclasse mais os campos e métodos declarados na própria subclasse (veja o rodapé na página 185 para explicações mais detalhadas). Desta forma, consideramos que a classe `ChefeDeDepartamento` pode acessar, direta ou indiretamente, os campos `departamento` e `promoçãoAChefe` (declarados na própria classe), `admissão` e `salário` (herdados da classe `Funcionario`) e `nome`, `identidade` e `nascimento` (herdados da classe `Pessoa`).
- O mecanismo de herança funciona somente em um sentido: da classe ancestral para a classe estendida, significando que a classe `PacienteDeClinica` terá acesso direto ou indireto aos campos da classe `Pessoa`, mas a classe `Pessoa` não terá acesso aos campos e métodos únicos da classe `PacienteDeClinica`, como `planoDeSaúde`.
- Similarmente, não existem relações entre duas classes estendidas que herdam de uma única classe ancestral, exceto os campos em comum que o mecanismo de herança propicia. Desta forma, a classe `Funcionario` não terá acesso ao campo `planoDeSaúde` da classe `PacienteDeClinica`.

*herança múltipla*

- Não existe um mecanismo de implementação de herança múltipla em Java, isto é, um mecanismo que faça com que uma classe herde métodos e campos de duas classes simultaneamente. No capítulo 9 veremos que é possível simular herança múltipla usando *interfaces*.

*A classe* `Object`

Todas as classes de Java descendem de uma classe chamada `Object` - mesmo que a declaração `extends Object` seja omitida de classes criadas pelo usuário, elas implicitamente herdarão da classe `Object`. Esta classe por si não contém campos ou métodos úteis, não devendo ser usada diretamente, servindo mais para declarar métodos genéricos (como, por exemplo, `toString`) que todas as classes devem implementar através da sobreposição (veja a seção 8.3.2).

### 8.3.1 A palavra-chave `super`

Nos exemplos mostrados nas classes `Funcionario` e `ChefeDeDepartamento` (listagens 8.8 e 8.9) vimos que classes derivadas ou subclasses podem ter acesso a métodos das superclasses, usando a palavra-chave `super`. O acesso a métodos de classes ancestrais é útil para aumentar a reutilização de código: se existem métodos na classe ancestral que podem efetuar parte do processamento necessário, devemos usar o código que já existe (e, esperançosamente, funciona) em vez de reescrevê-lo.

Existem duas maneiras de se reutilizar métodos de classes ancestrais que não tenham sido declarados como `private`: se a execução do método for a mesma para a superclasse e a subclasse, então instâncias da subclasse podem chamar diretamente o método como se fosse delas mesmas - é o caso do método `qualSalário`, que é declarado na classe `Funcionario` e pode ser executado tanto por instâncias da classe `Funcionario` quanto por instâncias da classe `ChefeDeDepartamento`, uma vez que esta classe herda da classe `Funcionario`. Como o método não teve que ser reescrito ou mesmo declarado na classe `ChefeDeDepartamento`, houve uma economia no desenvolvimento da classe descendente.

A segunda maneira de executar métodos da classe ancestral é mais complexa: partimos do pressuposto que não existe na classe ancestral um método que faça, para a classe descendente, exatamente o que queremos, mas existem métodos que executam parte da tarefa ou resolvem parte do problema. Desta maneira, uma classe descendente poderia executar a parte que resolve o problema parcial chamando o método correspondente da classe ancestral e depois executar comandos que completassem a função desejada.

Isto frequentemente ocorre em construtores: em muitos casos a função dos construtores é inicializar os campos das classes. Consideremos como exemplo a classe `Ponto2D`, que contém os campos `x` e `y` e seu construtor, que tem como finalidade inicializar estes campos com argumentos passados ao construtor. Consideremos a classe `Ponto3D` que herda da classe `Ponto2D` e contém o campo `z` - a classe `Ponto3D` também contém (via herança) os campos `x` e `y`. Se o construtor de `Ponto3D` for inicializar os campos `x`, `y` e `z`, ele pode delegar a inicialização de `x` e `y` ao construtor da classe `Ponto2D` e inicializar somente o campo `z`.

`super`

Um método ou construtor de uma subclasse pode chamar um método ou construtor de uma superclasse usando a palavra-chave `super`, cujo funcionamento é similar ao da palavra-chave `this` (seção 4.3.1) exceto que `super` se refere à classe ancestral imediata, e `this` à própria classe. Algumas regras para uso da palavra-chave `super` para chamar métodos de classes ancestrais como subrotinas são:

`super` *e construtores*

- Construtores são chamados simplesmente pela palavra-chave `super` seguida dos argumentos a serem passados para o construtor entre parênteses. Se não houverem argumentos, a chamada deve ser feita como `super()`.

`super` *e métodos*

- Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método. Se houverem argumentos a serem passados para o método, estes devem estar entre parênteses, após o nome do método, caso contrário (chamada de métodos que não recebem argumentos) os parênteses devem estar vazios.
- Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses, e mesmo assim, somente se forem declarados na primeira linha de código do construtor da subclasse (comentários não são considerados como código). Em outras palavras, quando um construtor de uma superclasse for chamado de dentro de um construtor de uma subclasse, a linha que faz a chamada deve ser a primeira do corpo do construtor da subclasse, e métodos não podem chamar construtores de superclasses.

*chamando construtores em cascata*

- Somente os métodos e construtores da superclasse imediata podem ser chamados usando a palavra-chave `super` - não existem construções como `super.super` que permitam a execução de métodos e construtores de classes ancestrais da classe ancestral.
  Caso seja necessário executar o construtor de uma classe ancestral da própria classe ancestral, os construtores podem ser escritos em *cascata*, de forma que se a classe `C` herda da classe `B` que por sua vez herda da classe `A`, o construtor da classe `C` pode chamar o construtor da classe `B` que por sua vez pode chamar o construtor da classe `A`. Um exemplo desta técnica é mostrado nos construtores das classes `ChefeDeDepartamento` e `Funcionario` (listagens 8.9 e 8.8).
- Se um método de uma classe ancestral for herdado pela classe descendente, ele pode ser chamado diretamente sem necessidade da palavra `super`. Por exemplo, se a classe `Robo` tem um método `andaParaAFrente` e a classe `MiniRobo` herda da classe `Robo`, a não ser que a classe `MiniRobo` sobreponha o método `andaParaAFrente` da classe `Robo`, ela poderá chamá-lo como se tivesse sido declarado na própria classe `MiniRobo`. O conceito de sobreposição será visto na seção 8.3.2.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

*execução implícita de construtores*

Um ponto particularmente curioso (e que causa confusão) é que o construtor de uma subclasse **sempre** chama o construtor de sua superclasse, mesmo que a chamada não seja explícita. Se a classe `Ponto3D` herda da classe `Ponto2D` o construtor de `Ponto3D` **deve** chamar o construtor da classe `Ponto2D`. Se a chamada não for explícita (através da palavra-chave `super`), o construtor da classe `Ponto3D` tentará chamar o construtor vazio (isto é, sem argumentos) da classe `Ponto2D` - se este construtor não estiver definido, um erro de compilação ocorrerá.

Desta forma é sempre importante lembrar que se uma classe não tem um construtor sem argumentos mas tem um com argumentos e uma classe herdeira é criada, esta classe herdeira deverá obrigatoriamente chamar o construtor com argumentos da classe ancestral.

Um exemplo mais detalhado que ilustra os tópicos discutidos anteriormente é dado pelas classes `Automovel`, `AutomovelBasico` e `AutomovelDeLuxo`, mostradas respectivamente nas listagens 8.10, 8.11 e 8.12.

A classe `Automovel` representa um automóvel à venda, com campos que descrevem os seus dados e métodos para manipulação destes dados. Consideraremos que um automóvel representado por esta classe não tem nenhum acessório ou equipamento adicional em especial. As classes `AutomovelBasico` e `AutomovelDeLuxo` encapsulam dados que representam automóveis com acessórios e equipamentos, além dos dados básicos encapsulados pela classe `Automovel`.

Listagem 8.10: A classe `Automovel`, que encapsula os dados de um automóvel simples à venda.

```
1  /**
2   * A classe Automovel, que encapsula os dados de um automóvel à venda.
3   */
4  class Automovel
5    {
6   /**
7    * Declaração de alguns campos constantes desta classe. Estes campos são declarados
8    * como public para serem acessíveis de fora da classe, como static pois serão
9    * independentes de instâncias, e como final para que não sejam modificáveis.
10   */
11   public static final byte MOVIDOAGASOLINA = 1;
12   public static final byte MOVIDOAALCOOL   = 2;
13   public static final byte MOVIDOADIESEL   = 3;
14   public static final byte MOVIDOAGAS      = 4;
15   // Este valor representa o número máximo de prestações, e é usado somente para
16   // possibilitar cálculos de financiamento.
17   private static final byte NÚMEROMÁXIMODEPRESTAÇÕES = 24;
18
19  /**
20   * Declaração dos campos da classe
21   */
22   private String modelo;        // o modelo do automóvel
23   private String cor;           // a cor do automóvel
24   private byte combustível;     // o tipo de combustível do automóvel, veja lista
25                                 // declarada anteriormente nesta classe
26
27  /**
28   * O construtor da classe Automovel, que recebe argumentos para inicializar
29   * seus campos.
30   * @param m o modelo do automóvel
31   * @param c a cor do automóvel
32   * @param comb o tipo de combustível que o automóvel usa
33   */
34   Automovel(String m,String c,byte comb)
35     {
36     modelo = m;
37     cor = c;
38     combustível = comb;
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
39      }
40
41    /**
42     * O método quantoCusta retorna o preço do automóvel, calculado de acordo com o
43     * tipo de combustível.
44     * @return o preço do automóvel
45     */
46     public float quantoCusta()
47       {
48       float preço = 0; // a variável deve ser inicializada
49       switch(combustível)
50         {
51         case MOVIDOAGASOLINA: preço = 12000.0f; break; // Notem o sufixo 'f' ao final
52         case MOVIDOAALCOOL:   preço = 10500.0f; break; // de cada valor para denotar
53         case MOVIDOADIESEL:   preço = 11000.0f; break; // um valor do tipo float e não
54         case MOVIDOAGAS:      preço = 13000.0f; break; // double (que seria o default)
55         }
56       return preço;
57       }
58
59    /**
60     * O método quantasPrestações retorna o número máximo de prestações para este tipo
61     * de automóvel.
62     * @return o número máximo de prestações
63     */
64     public byte quantasPrestações()
65       {
66       return NÚMEROMÁXIMODEPRESTAÇÕES;
67       }
68
69    /**
70     * O método toString retorna uma String contendo os campos desta classe com uma
71     * formatação simples.
72     * @return os campos da classe, formatados
73     */
74     public String toString()
75       {
76       String resultado; // a String que irá receber a concatenação dos campos
77       resultado = modelo+" "+cor+"\n"; // inicializamos o resultado com o modelo e cor
78       switch(combustível) // adicionamos a descrição do combustível
79         {
80         case MOVIDOAGASOLINA: resultado += "Movido a Gasolina\n"; break;
81         case MOVIDOAALCOOL:   resultado += "Movido a Álcool\n"; break;
82         case MOVIDOADIESEL:   resultado += "Movido a Diesel\n"; break;
83         case MOVIDOAGAS:      resultado += "Movido a Gás\n"; break;
84         }
85       return resultado;  // retorna o resultado da concatenação
86       }
87
88    } // fim da classe Automovel
```

Alguns pontos interessantes da classe `Automovel` (listagem 8.10) são:

- Vemos novamente o uso de campos declarados como `public static final` para representar constantes que podem ser usadas por métodos da classe e por aplicações ou outras classes que a utilizem.
- Um campo da classe que representa o número máximo de prestações para este tipo de automóvel é declarado como `private static final` - este valor, que não poderá ser acessado diretamente (`private`), será constante para todas as instâncias desta classe (`static`) e não poderá ser modificado por métodos desta classe ou de classes herdeiras (`final`) somente poderá ser acessado através do método `quantasPrestações`. Os efeitos de campos finais e estáticos serão vistos nos exemplos a seguir.

A classe `Automovel` é usada como base para a criação da classe `AutomovelBasico`, que além de conter os campos e métodos não-privados de `Automovel` ainda tem campos que represen-



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

tam acessórios e opcionais de um automóvel. A classe `AutomovelBasico` é mostrada na listagem 8.11.

Listagem 8.11: A classe `AutomovelBasico`, que encapsula os dados de um automóvel básico à venda, e que herda da classe `Automovel`.

```
/**
 * A classe AutomovelBasico, que encapsula os dados de um automóvel básico à venda, e
 * que herda da classe Automovel.
 */
class AutomovelBasico extends Automovel
  {
 /**
  * Declaração dos campos da classe (acessórios que um automóvel básico pode ter)
  */
  private boolean retrovisorDoLadoDoPassageiro;
  private boolean limpadorDoVidroTraseiro;
  private boolean rádioAMFM;

 /**
  * O construtor da classe AutomovelBasico, que recebe argumentos para inicializar
  * seus campos.
  * @param m o modelo do automóvel
  * @param c a cor do automóvel
  * @param comb o tipo de combustível que o automóvel usa
  * @param retro true se o automóvel tem retrovisor do lado do passageiro
  * @param limpa true se o automóvel tem limpador do vidro traseiro
  * @param rádio true se o automóvel tem rádio AM/FM
  */
  AutomovelBasico(String m,String c,byte comb,
                  boolean retro,boolean limpa,boolean rádio)
    {
    super(m,c,comb); // A inicialização dos campos herdados é feita pela superclasse.
    retrovisorDoLadoDoPassageiro = retro; // a inicialização dos campos desta classe
    limpadorDoVidroTraseiro = limpa;      // é feita diretamente pelo construtor
    rádioAMFM = rádio;
    }

 /**
  * O construtor da classe AutomovelBasico, que recebe argumentos para inicializar
  * seus campos. Esta versão do construtor considera que os acessórios estão todos
  * presentes.
  * @param m o modelo do automóvel
  * @param c a cor do automóvel
  * @param comb o tipo de combustível que o automóvel usa
  */
  AutomovelBasico(String m,String c,byte comb)
    {
    super(m,c,comb); // a inicialização dos campos herdados é feita pela superclasse
    retrovisorDoLadoDoPassageiro = true;
    limpadorDoVidroTraseiro = true;
    rádioAMFM = true;
    }

 /**
  * O método quantoCusta retorna o preço do automóvel, calculado de acordo com o
  * tipo de combustível. Este método usa o método quantoCusta da superclasse para
  * calcular o preço-base do automóvel, e acrescenta o valor dos opcionais.
  * @return o preço do automóvel
  */
  public float quantoCusta()
    {
    float preço = super.quantoCusta(); // calcula o preço-base do automóvel
    if (retrovisorDoLadoDoPassageiro) preço += 280;
    if (limpadorDoVidroTraseiro)      preço += 650;
    if (rádioAMFM)                    preço += 190;
    return preço;
```

```
62      }
63
64    /**
65     * O método toString retorna uma String contendo os campos desta classe com uma
66     * formatação simples.
67     * @return os campos da classe, formatados
68     */
69    public String toString()
70      {
71      String resultado = super.toString(); // a String que irá receber a concatenação
72                                           // dos campos, já recebendo os dados
73                                           // formatados do automóvel
74      if (retrovisorDoLadoDoPassageiro)
75        resultado +=  "Com retrovisor do lado do passageiro\n";
76      if (limpadorDoVidroTraseiro)
77        resultado +=  "Com limpador do vidro traseiro\n";
78      if (rádioAMFM)
79        resultado +=  "Com rádio AM/FM\n";
80      return resultado;  // retorna o resultado da concatenação
81      }
82
83    } // fim da classe AutomovelBasico
```

Algumas características importantes da classe `AutomovelBasico` (listagem 8.11) são:

- A primeira instrução dos construtores da classe `AutomovelBasico` (existem dois, graças à sobrecarga) executam o construtor da classe imediatamente ancestral à ela, ou seja, o construtor da classe `Automovel`, repassando ao construtor da classe ancestral alguns dos argumentos recebidos - os argumentos correspondentes aos campos herdados da classe `Automovel`.
  Note que a execução dos construtores são feitas pela palavra-chave `super` sem nenhum modificador ou nome de método ou classe adicional.
- O método `quantoCusta` da classe `AutomovelBasico` *sobrepõe* o método com a mesma assinatura que foi herdado da classe `Automovel`. O conceito e as regras de sobreposição de métodos herdados serão vistos com mais detalhes na seção 8.3.2. Este método executa o método de mesmo nome da classe ancestral chamando-o com a palavra-chave `super`, na forma `super.quantoCusta`. O valor calculado pelo método da classe ancestral é acrescido dos valores relacionados aos campos encapsulados pela classe `AutomovelBasico` (correspondentes aos acessórios do automóvel).
- O método `toString` da classe `AutomovelBasico` segue praticamente as mesmas regras do método `quantoCusta` - executa o método da superclasse usando a palavra-chave `super`, armazenando o resultado e concatenando este resultado com dados específicos da classe `AutomovelBasico`.
- A classe `AutomovelBasico` não tem declaração para o método `quantasPrestações` nem redeclara o campo privado `NÚMEROMÁXIMODEPRESTAÇÕES` - apesar do campo ser privado, o método que permite o seu acesso é público, e é herdado diretamente da classe `Automovel`, significando que o método pode ser chamado diretamente de qualquer instância da classe `AutomovelBasico` ou de dentro de qualquer método desta classe, *sem* a necessidade do uso da palavra-chave `super`. Para os efeitos práticos, todas as instâncias da classe `AutomovelBasico` representarão automóveis que podem ser pagos em até 24 prestações (valor definido para o campo privado `NÚMEROMÁXIMODEPRESTAÇÕES` na classe `Automovel`).

Para reforçar mais ainda os conceitos de herança e uso da palavra-chave `super`, consideremos a classe `AutomovelDeLuxo`, que herda diretamente da classe `AutomovelBasico` e indiretamente da classe `Automovel`. A classe `AutomovelDeLuxo` é mostrada na listagem 8.12.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem 8.12: A classe `AutomovelDeLuxo`, que encapsula os dados de um automóvel de luxo à venda, e que herda da classe `AutomovelBasico`.

```
1  /**
2   * A classe AutomovelDeLuxo, que encapsula os dados de um automóvel básico à venda, e
3   * que herda da classe AutomovelBasico.
4   */
5  class AutomovelDeLuxo extends AutomovelBasico
6    {
7    // Este valor representa o número máximo de prestações, e é usado somente para
8    // possibilitar cálculos de financiamento.
9    private static final byte NÚMEROMÁXIMODEPRESTAÇÕES = 36;
10
11   /**
12    * Declaração dos campos da classe (acessórios que um automóvel de luxo pode ter)
13    */
14   private boolean direçãoHidráulica;
15   private boolean câmbioAutomático;
16   private boolean vidrosETravasElétricos;
17
18   /**
19    * O construtor da classe AutomovelDeLuxo, que recebe argumentos para inicializar
20    * seus campos.
21    * @param m o modelo do automóvel
22    * @param c a cor do automóvel
23    * @param comb o tipo de combustível que o automóvel usa
24    * @param retro true se o automóvel tem retrovisor do lado do passageiro
25    * @param limpa true se o automóvel tem limpador do vidro traseiro
26    * @param rádio true se o automóvel tem rádio AM/FM
27    * @param dir   true se o automóvel tem direção hidráulica
28    * @param camb  true se o automóvel tem câmbio automático
29    * @param vidro true se o automóvel tem vidros e travas elétricos
30    */
31   AutomovelDeLuxo(String m,String c,byte comb,
32                   boolean retro,boolean limpa,boolean rádio,
33                   boolean dir,boolean camb,boolean vidro)
34     {
35     super(m,c,comb,retro,limpa,rádio); // a inicialização dos campos herdados é
36                                        // feita pela superclasse
37     direçãoHidráulica = dir; // a inicialização dos campos desta classe é feita
38     câmbioAutomático = camb; // diretamente pelo construtor
39     vidrosETravasElétricos = vidro;
40     }
41
42   /**
43    * O construtor da classe AutomovelDeLuxo, que recebe argumentos para inicializar
44    * seus campos. Esta versão do construtor considera que os acessórios estão todos
45    * presentes.
46    * @param m o modelo do automóvel
47    * @param c a cor do automóvel
48    * @param comb o tipo de combustível que o automóvel usa
49    */
50   AutomovelDeLuxo(String m,String c,byte comb)
51     {
52     super(m,c,comb); // a inicialização dos campos herdados é feita pela superclasse
53     direçãoHidráulica = true;
54     câmbioAutomático = true;
55     vidrosETravasElétricos = true;
56     }
57
58   /**
59    * O método quantoCusta retorna o preço do automóvel, calculado de acordo com o
60    * tipo de combustível. Este método usa o método quantoCusta da superclasse para
61    * calcular o preço-base do automóvel, e acrescenta o valor dos opcionais.
62    * @return o preço do automóvel
63    */
64   public float quantoCusta()
65     {
```

```
66      float preço = super.quantoCusta(); // calcula o preço-base do automóvel
67      if (direçãoHidráulica) preço      += 5340;
68      if (câmbioAutomático)  preço      += 7500;
69      if (vidrosETravasElétricos) preço += 2320;
70      return preço;
71      }
72
73    /**
74     * O método quantasPrestações retorna o número máximo de prestações para este tipo
75     * de automóvel. É necessário sobrepor o método herdado da classe Automóvel para
76     * que o valor correto seja impresso !
77     * @return o número máximo de prestações
78     */
79    public byte quantasPrestações()
80      {
81      return NÚMEROMÁXIMODEPRESTAÇÕES;
82      }
83
84    /**
85     * O método toString retorna uma String contendo os campos desta classe com uma
86     * formatação simples.
87     * @return os campos da classe, formatados
88     */
89    public String toString()
90      {
91      String resultado = super.toString(); // a String que irá receber a concatenação
92                                           // dos campos, já recebendo os dados
93                                           // formatados do automóvel
94      if (direçãoHidráulica)
95        resultado +=  "Com direção hidráulica\n";
96      if (câmbioAutomático)
97        resultado +=  "Com câmbio automático\n";
98      if (vidrosETravasElétricos)
99        resultado +=  "Com vidros e travas elétricas\n";
100     return resultado;  // retorna o resultado da concatenação
101     }
102
103   } // fim da classe AutomovelDeLuxo
```

Alguns pontos interessantes da classe `AutomovelDeLuxo` são descritos a seguir:

*exemplo de construtores em cascata*

- Novamente a primeira instrução do construtor da classe `AutomovelDeLuxo` executa o construtor da classe imediatamente ancestral à ela (`AutomovelBasico`), que por sua vez executará o construtor da sua classe ancestral, `Automovel` - os construtores das classes são chamados em *cascata*, cada um chamando por sua vez o construtor da classe imediatamente superior.

*redeclaração de campos* `final`

- O campo privado `NÚMEROMÁXIMODEPRESTAÇÕES` é redeclarado nesta classe, sobrepondo a declaração de um campo com o mesmo nome e modificadores de acesso declarado na classe `Automovel` e herdado pela classe `AutomovelBasico` - em outras palavras, as instâncias da classe `AutomovelDeLuxo` terão 36 como valor do campo `NÚMEROMÁXIMODEPRESTAÇÕES`. O campo foi declarado como `final` em classes ancestrais, portanto o seu valor não pode ser modificado, entretanto, o campo pode ser redeclarado ou sobreposto.
  É importante notar que o método `quantasPrestações` também foi redeclarado - caso não tivesse sido, o método `quantasPrestações` da classe `AutomovelDeLuxo` seria o herdado da classe `Automovel`, que retornaria o valor do campo `NÚMEROMÁXIMODEPRESTAÇÕES` da própria classe `Automovel`, dando um resultado diferente do esperado.
- O método `quantoCusta` da classe `AutomovelDeLuxo` também sobrepõe o método com a mesma assinatura que foi herdado da classe `AutomovelBasico`. Este método executa o método de mesmo nome da classe ancestral imediata, `AutomovelBasico`, que por sua vez chama o método de mesmo nome da sua classe ancestral, `Automovel`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

- O método `toString` da classe `AutomovelDeLuxo` também executa o método de mesmo nome da superclasse usando a palavra-chave `super`, armazenando o resultado e concatenando este resultado com dados específicos da classe `AutomovelDeLuxo`. Veja também a descrição do método `toString` da classe `AutomovelBasico`.

A classe `DemoAutomoveis`, mostrada na listagem 8.13, exemplifica o uso de instâncias das classes `Automovel`, `AutomovelBasico` e `AutomovelDeLuxo`.

Listagem 8.13: A classe `DemoAutomoveis`, que demonstra instâncias das classes `Automovel`, `AutomovelBasico` e `AutomovelDeLuxo`.

```
1  /**
2   * A classe DemoAutomoveis, que demonstra instâncias das classes Automovel,
3   * AutomovelBasico e AutomovelDeLuxo.
4   */
5  class DemoAutomoveis
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * algumas instâncias das classes Automovel, AutomovelBasico e AutomovelDeLuxo, e
10   * chamadas a métodos destas classes.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Criamos uma instância da classe Automovel, cujo construtor somente recebe como
17     // argumentos o modelo, a cor e o combustível. Note como podemos usar as
18     // constantes finais declaradas na classe Automovel como mnemônicos para o tipo
19     // de combustível.
20     Automovel a = new Automovel("Fusca","verde",Automovel.MOVIDOAALCOOL);
21     // Executamos alguns métodos da classe Automovel
22     System.out.println(a); // toString (implicitamente)
23     System.out.println(a.quantoCusta());
24     System.out.println(a.quantasPrestações());
25
26     // Criamos uma instância da classe AutomovelBasico. Usamos o construtor que recebe
27     // como argumentos o modelo, a cor e o combustível (que serão repassados para o
28     // construtor da classe ancestral), além de booleanos para representar a existência
29     // de retrovisores do lado do passageiro, limpador de pára-brisas e rádio AM/FM.
30     // A mesma classe tem um construtor que considera estes três acessórios como
31     // existentes por default.
32     AutomovelBasico ab = new AutomovelBasico("Corsa","cinza",Automovel.MOVIDOAGASOLINA,
33                                              true,true,false);
34     // Executamos alguns métodos da classe AutomovelBasico
35     System.out.println(ab); // toString (implicitamente)
36     System.out.println(ab.quantoCusta());
37     System.out.println(ab.quantasPrestações()); // método herdado, sem modificações
38
39     // Criamos uma instância da classe AutomovelDeLuxo. Usamos o construtor que recebe
40     // como argumentos somente o modelo, a cor e o combustível (que serão repassados
41     // para o construtor da classe ancestral, em cascata). Este construtor considera
42     // que todos os acessórios são instalados por default. A mesma classe tem um
43     // construtor que recebe valores booleanos indicando a existência ou não de cada
44     // um dos acessórios.
45     AutomovelDeLuxo al = new AutomovelDeLuxo("Classe A","azul",
46                                              Automovel.MOVIDOAGASOLINA);
47     // Executamos alguns métodos da classe AutomovelDeLuxo
48     System.out.println(al); // toString (implicitamente)
49     System.out.println(al.quantoCusta());
50     System.out.println(al.quantasPrestações()); // método superposto
51     }
52
53   } // fim da classe DemoAutomoveis
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

### 8.3.2 Sobreposição e ocultação

Vimos nos exemplos mostrados neste capítulo que quando criamos classes estendidas através do mecanismo de herança, podemos definir novos campos e métodos que são específicos da classe herdeira (como visto nas listagens 8.7, 8.8 e 8.9) mas também redeclarar campos e métodos que tem a mesma assinatura de métodos declarados em classes ancestrais, mas com funcionalidade diferente (como nas listagens 8.10, 8.11 e 8.12).

*sobreposição*
*superposição*

A declaração de métodos com a mesma assinatura que métodos de classes ancestrais chama-se *sobreposição* ou *superposição*. A razão de sobrepormos métodos é que métodos de classes herdeiras geralmente executam tarefas adicionais que os mesmos métodos das classes ancestrais não executam.

O exemplo mais óbvio da necessidade de sobreposição é dado por métodos que fazem a impressão de campos: a classe ancestral somente imprime os campos que contém, então se adicionamos campos à classe descendente, o método de impressão desta não poderá ser o herdado, que não imprimirá os campos adicionais.

*ocultação de campos*

A declaração de campos em uma classe descendente com o mesmo nome de campos declarados na classe ancestral chama-se *ocultação*. Ao contrário da sobreposição de métodos, que é bastante útil e comum em classes herdeiras, a ocultação de campos não oferece muitas vantagens, e as poucas oferecidas podem facilmente ser implementadas através de métodos que retornam valores e são superpostos de acordo com a necessidade.

As principais regras de sobreposição de métodos e ocultação de campos, e regras de uso de modificadores de acesso em classes herdadas são:

*regras de superposição*

- A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral - este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`. Veja, como exemplo, os vários métodos `toString` de classes herdeiras mostradas neste capítulo.
- Métodos declarados em uma subclasse com o mesmo nome mas assinaturas diferentes (por exemplo, número de argumentos diferentes) dos métodos da superclasse não sobrepõem estes métodos. Por exemplo, se a classe `Cor` conter o método não-privado `inicializa` que recebe três valores do tipo `float` como argumentos, e a classe `CorCMYK` conter o método com mesmo nome mas que recebe quatro valores do tipo `float`, esta classe poderá acessar diretamente o método `inicializa` da classe ancestral pois as assinaturas dos métodos de mesmo nome serão diferentes - em outras palavras, o método `inicializa` será herdado como qualquer outro.
- Métodos podem ser sobrepostos com diferentes modificadores de acesso, contanto que os métodos sobrepostos tenham modificadores de acesso menos restritivos. Em outras palavras, podemos declarar um método na superclasse com o modificador de acesso `private` e sobrepor este método em uma subclasse com o modificador de acesso `public`, mas não podemos fazer o contrário.
- Métodos estáticos declarados em classes ancestrais não podem ser sobrepostos em classes descendentes, nem mesmo se não forem declarados como estáticos.
- Se um campo é declarado em uma superclasse e oculto em subclasses, e métodos que acessam este campo são herdados, estes métodos farão referência ao campo da classe onde foram declarados.
  Um exemplo é visto nas classes que representam diferentes tipos de automóveis, vistas



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

neste capítulo. O campo `NÚMEROMÁXIMODEPRESTAÇÕES` e o método `quantasPrestações`, que retorna o valor deste campo, foram declarados na classe `Automovel` (listagem 8.10) e herdados diretamente pela classe `AutomovelBasico` (listagem 8.11). O campo é redeclarado (oculto) pela classe `AutomovelDeLuxo` (listagem 8.12), que redeclara também o método `quantasPrestações`. Se este método não fosse redeclarado, sobrepondo o da classe ancestral, seria herdado, mas retornaria o valor do campo declarado na superclasse. Em outras palavras, métodos herdados não podem acessar campos declarados em subclasses.

- Qualquer método da classe herdeira pode chamar qualquer método da classe ancestral que tenha sido declarado como `public`, `protected` ou sem declaração explícita de modificador. Métodos declarados como `private` não são acessíveis diretamente, mas podem ser chamados indiretamente a partir de métodos que não sejam `private`.
  Por exemplo, consideremos a classe `ComputadorDeBordo`, que encapsula os campos e métodos necessários para implementar um computador de bordo de automóveis. Esta classe poderia ter o método privado `calculaQuilometragemRodada` e o método público `mostraEstatísticasDeConsumo`, com o método `mostraEstatísticasDeConsumo` executando o método `calculaQuilometragemRodada` como parte de seu processamento. Se a classe `ComputadorDeBordoDeCaminhao` for criada como sendo herdeira da classe `ComputadorDeBordo`, ela não poderá executar o método `calculaQuilometragemRodada` diretamente pois este foi declarado como sendo privado da classe `ComputadorDeBordo`, mas o método será executado indiretamente através da chamada ao método público herdado `mostraEstatísticasDeConsumo`.

*métodos* `final`

- Métodos declarados como `final` são herdados por subclasses, mas não podem ser sobrepostos (a não ser que a sua assinatura seja diferente). Por exemplo, a classe `ChefeDeDepartamento` (listagem 8.9) não pode declarar um método `qualSalário` pois este foi declarado como final na classe ancestral `Funcionario` (listagem 8.8).
- As regras de acesso e herança envolvendo o modificador `protected` serão vistas com mais detalhes no capítulo 10.

*classes declaradas como* `final`

Classes inteiras podem ser declaradas como finais, na forma `final class ...`. Se uma classe é declarada como final, todos os seus métodos serão finais, mas não os seus campos. A declaração de uma classe como final efetivamente impede o mecanismo de herança - o compilador não compilará uma classe declarada como herdeira de uma classe final.

### 8.3.3 Polimorfismo

O mecanismo de herança permite a criação de classes a partir de outras já existentes com relações *é-um-tipo-de*, de forma que a partir de uma classe genérica, classes mais especializadas possam ser criadas. Vimos, nesta seção, alguns exemplos de classes que seguem esta regra: `Funcionario` *é-um-tipo-de* `Pessoa`, e `AutomovelDeLuxo` *é-um-tipo-de* `AutomovelBasico`, que por sua vez *é-um-tipo-de* `Automovel`.

*polimorfismo*

A relação *é-um-tipo-de* entre classes permite a existência de outra característica fundamental de linguagens de programação orientadas a objetos: *polimorfismo*. Polimorfismo ("muitas formas") permite a manipulação de instâncias de classes que herdam de uma mesma classe ancestral de forma unificada: podemos escrever métodos que recebam instâncias de uma classe `C`, e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe `C`, já que qualquer classe que herde de `C` *é-um-tipo-de* `C`. Um exemplo prático de polimorfismo é dado pela classe `ConcessionariaDeAutomoveis`, mostrada na listagem 8.14.

Listagem 8.14: A classe `ConcessionariaDeAutomoveis`, que demonstra polimorfismo com instâncias das classes herdeiras da classe `Automovel`.

```
/**
 * A classe ConcessionariaDeAutomoveis, que implementa o processo de venda de
 * automóveis em uma concessionária. Para cada tipo de automóvel, o preço e número de
 * suas prestações será impresso. O modelo representado é extremamente simples, e a
 * concessionária somente tem cinco automóveis à venda.
 * Esta classe demonstra polimorfismo de classes usando instâncias das classes
 * herdeiras da classe Automovel. Um método da classe aceitará instâncias de qualquer
 * classe herdeira de Automovel.
 */
class ConcessionariaDeAutomoveis
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * algumas instâncias das classes Automovel, AutomovelBasico e AutomovelDeLuxo, e as
  * usa para mostrar o valor da prestação de cada automóvel.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Criamos uma instância da classe Automovel
    Automovel a1 = new Automovel("Fiat","bege",Automovel.MOVIDOAALCOOL);
    // Criamos duas instâncias da classe AutomovelBasico
    // Uma com o construtor básico
    AutomovelBasico a2 = new AutomovelBasico("Corsa","cinza",
                                             Automovel.MOVIDOAGASOLINA);
    // E outra com o construtor mais completo
    AutomovelBasico a3 = new AutomovelBasico("Gol","branco",Automovel.MOVIDOAGASOLINA,
                                             false,false,true);
    // Criamos duas instâncias da classe AutomovelDeLuxo
    // Uma com o construtor básico
    AutomovelDeLuxo a4 = new AutomovelDeLuxo("Ibiza","vermelho",
                                             Automovel.MOVIDOAGASOLINA);
    // E outra com o construtor mais completo
    AutomovelDeLuxo a5 = new AutomovelDeLuxo("Honda","prata",Automovel.MOVIDOAGASOLINA,
                                             true,true,false,
                                             true,false,true);
    // Imprimimos o "catálogo" da concessionária, ou seja, os dados dos automóveis,
    // preços e condições de pagamento.
    imprime(a1);
    imprime(a2);
    imprime(a3);
    imprime(a4);
    imprime(a5);
    } // fim do método main

 /**
  * O método imprime mostra os dados de uma instância de qualquer classe que
  * descenda da classe Automovel. Os métodos quantoCusta, quantasPrestações e
  * toString das instâncias serão chamados.
  * @param a uma instância de qualquer classe que herde da classe Automovel.
  */
  public static void imprime(Automovel a)
    {
    System.out.println("Seguem os dados do automóvel escolhido:");
    System.out.print(a); // chamada implícita a toString
    System.out.println("Valor: "+a.quantoCusta());
    System.out.println(a.quantasPrestações()+" prestações de "+
                       (a.quantoCusta()/a.quantasPrestações()));
    } // fim do método imprime

  } // fim da classe ConcessionariaDeAutomoveis
```

A classe `ConcessionariaDeAutomoveis` tem um método (`imprime`) que recebe uma instância da classe `Automovel` como argumento, e imprime os dados desta instância, usando os métodos



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

`toString`, `quantoCusta` e `quantasPrestacoes` da classe. Este mesmo método pode receber, como argumentos, instâncias das classes `AutomovelBasico`, `AutomovelDeLuxo` e de qualquer outra classe que herde da classe `Automovel` ou de suas herdeiras, uma vez que estas terão os métodos necessários para a execução do método `imprime`.

Vale a pena notar que é possível implementarmos, em uma classe, métodos polimórficos e sobrecarregados simultaneamente. Poderíamos, por exemplo, criar métodos para processamento de instâncias de uma classe genérica e outros que processariam instâncias de classes mais especializadas (subclasses) de forma diferenciada. A classe `EmprestimoBancario`, na listagem 8.15, demonstra uma aplicação desta técnica.

Listagem 8.15: A classe `EmprestimoBancario`, que demonstra polimorfismo com instâncias das classes herdeiras da classe `Pessoa`.

```
/**
 * A classe EmprestimoBancario, que calcula o valor de empréstimos que podem ser dados
 * a pessoas (instâncias das classes Pessoa, Funcionario e ChefeDeDepartamento) por um
 * banco popular. Empréstimos a pessoas que não são funcionárias são de valor fixo,
 * enquanto empréstimos dados a funcionários e chefes de departamento são baseados
 * nos salários que estes recebem.
 * Esta classe demonstra polimorfismo de classes usando instâncias das classes
 * herdeiras da classe Pessoa, e sobrecarga, implementando o mesmo método mas mais
 * especializado para instâncias de classes herdeiras da classe Funcionario.
 */
class EmprestimoBancario
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * algumas instâncias das classes Pessoa, Funcionario e ChefeDeDepartamento, e as
  * usa para calcular o valor de um empréstimo bancário a ser feito para cada uma
  * das pessoas representadas pela instância.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Criamos uma instância da classe Pessoa
    Pessoa p1 = new Pessoa("Kurt Gödel",10973213,
                           new Data((byte)23,(byte)12,(short)1904));
    // Criamos três instâncias da classe Funcionario
    Funcionario f1 = new Funcionario("Henri Poincaré",19283712,
                                     new Data((byte)12,(byte)7,(short)1897),
                                     new Data((byte)28,(byte)1,(short)1918),
                                     2500.0f);
    Funcionario f2 = new Funcionario("Paul Dirac",98736812,
                                     new Data((byte)20,(byte)1,(short)1885),
                                     new Data((byte)31,(byte)3,(short)1909),
                                     3200.0f);
    Funcionario f3 = new Funcionario("Wolfgang Pauli",33886620,
                                     new Data((byte)14,(byte)9,(short)1902),
                                     new Data((byte)16,(byte)11,(short)1930),
                                     3600.0f);
    // Criamos uma instância da classe ChefeDeDepartamento
    ChefeDeDepartamento c1 = new ChefeDeDepartamento("Edwin Hubble",4259782,
                                                     new Data((byte)20,(byte)1,(short)1875),
                                                     new Data((byte)20,(byte)7,(short)1899),
                                                     4100.0f,
                                                     "Laboratório de Astrofísica",
                                                     new Data((byte)20,(byte)7,(short)1899));
    // Calculamos os empréstimos que podem ser feitos a cada pessoa
    System.out.println(calculaEmpréstimo(p1));
    System.out.println(calculaEmpréstimo(f1));
    System.out.println(calculaEmpréstimo(f2));
    System.out.println(calculaEmpréstimo(f3));
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
51      System.out.println(calculaEmpréstimo(c1));
52      } // fim do método main
53
54    /**
55     * O método calculaEmpréstimo calcula o empréstimo que pode ser dado para uma pessoa
56     * (instância da classe Pessoa ou de suas descendentes, mas que não sejam instâncias
57     * da classe Funcionario ou suas descendentes). Veja também o método sobrecarregado
58     * abaixo.
59     * @param p uma instância de qualquer classe que herde da classe Pessoa
60     */
61    public static float calculaEmpréstimo(Pessoa p)
62      {
63      return 1000.0f; // qualquer pessoa pode ter um empréstimo de 1000 reais.
64      } // fim do método calculaEmpréstimo
65
66    /**
67     * O método calculaEmpréstimo calcula o empréstimo que pode ser dado para uma
68     * pessoa (instância da classe Funcionario e suas descendentes). O empréstimo será
69     * calculado de forma diferenciada para as instâncias das classes Funcionario e
70     * ChefeDeDepartamento.
71     * @param f uma instância de qualquer classe que herde da classe Funcionario
72     */
73    public static float calculaEmpréstimo(Funcionario f)
74      {
75      float empréstimo = 0f; // inicialmente consideramos o empréstimo zero
76      // Primeiro verificamos se a instância f é uma instância da classe
77      // ChefeDeDepartamento. Se for, calculamos o empréstimo como sendo quatro vezes o
78      // salário recebido.
79      if (f instanceof ChefeDeDepartamento)
80        {
81        empréstimo = 4.0f*f.qualSalário();
82        }
83      // Se a instância f não for da classe ChefeDeDepartamento, verificaremos se ela é
84      // instância da classe Funcionario, e se for, calculamos o empréstimo como sendo
85      // duas vezes o salário recebido.
86      else if (f instanceof Funcionario)
87        {
88        empréstimo = 2.0f*f.qualSalário();
89        }
90      return empréstimo;
91      } // fim do método calculaEmpréstimo
92
93    } // fim da classe EmprestimoBancario
```

Os pontos interessantes da classe `EmprestimoBancario` são:

- O método `main` da classe `EmprestimoBancario` declara e inicializa uma instância da classe `Pessoa`, três da classe `Funcionario` e uma da classe `ChefeDeDepartamento`. Estas instâncias serão passadas como argumentos para o método `calculaEmpréstimo`, que tem uma versão que aceita instâncias da classe `Pessoa` e outra versão, sobrecarregada, que aceita instâncias da classe `Funcionario`.
- O interpretador Java decidirá, em tempo de execução, qual dos dois métodos será chamado para cada instância criada no método `main`. Se a instância passada como argumento for da classe `ChefeDeDepartamento` ou da classe `Funcionario`, o segundo método será chamado: apesar das classes `ChefeDeDepartamento` e `Funcionario` serem *um-tipo-de* `Pessoa`, elas são mais próximas da classe `Funcionario` do que da classe `Pessoa`. Em outras palavras, se houverem vários métodos sobrecarregados que tratam de classes herdeiras de uma classe ancestral, o interpretador Java tentará sempre chamar o método mais adequado para cada instância, sendo que o método mais adequado será o que receber como argumentos a classe mais próxima em termos de herança (A classe `ChefeDeDepartamento` é mais próxima da classe `Funcionario` do que da classe `Pessoa`).
- O primeiro método `calculaEmpréstimo` recebe como argumento uma instância da classe



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

`Pessoa` ou de qualquer classe herdeira desta que não seja herdeira da classe `Funcionario` (já que instâncias de classes herdeiras de `Funcionario` serão processadas pelo segundo método `calculaEmpréstimo`). Este método retorna um valor constante, assumindo que qualquer pessoa (que não um funcionário) possa ter um empréstimo de mil reais.
Se uma instância da classe `Aluno`, que herdasse da classe `Pessoa`, fosse passada como argumento para o método `calculaEmpréstimo`, a primeira versão seria chamada. Se por outro lado, uma instância da classe `PresidenteDaEmpresa`, que fosse herdeira da classe `ChefeDeDepartamento` fosse passada para o método, a segunda versão seria executada. Um exemplo gráfico de chamadas a métodos polimórficos sobrecarregados é mostrado na figura 8.2.

`instanceof`

- O segundo método `calculaEmpréstimo`, que calculará o empréstimo para qualquer instância de qualquer classe que herde da classe `Funcionario`, verifica internamente se a instância passada como argumento é uma instância da classe `ChefeDeDepartamento` ou da própria classe `Funcionario`. Isto é feito com a palavra-chave `instanceof`, em expressões como `referência-à-instância instanceof nome-da-classe`, que retorna `true` se `referência-à-instância` for uma instância de `nome-da-classe` e `false` se não o for.
  É importante notar que o método primeiramente verifica se a instância passada é uma instância da classe `ChefeDeDepartamento` (mais específica) para **depois** verificar se a instância é da classe `Funcionario` (mais genérica). A ordem de verificação é essencial, pois uma instância da classe `ChefeDeDepartamento` também é uma instância da classe `Funcionario` - se esta fosse verificada primeiro, a condição marcada com o `else` no método nunca seria verificada. Em outras palavras, todas as instâncias passadas para este método são da classe `Funcionario`, mas se algumas forem da classe `ChefeDeDepartamento` (ou de outras classes herdeiras de `Funcionario`), a verificação destas deve ser feita primeiro - a verificação de a que classe pertencem as instâncias deve ser feita da classe mais específica para a mais genérica.

cast *de instâncias de classes*

- O método `qualSalário` é definido na classe `Funcionario` e nas suas herdeiras, mas não na classe `Pessoa`, ancestral de `Funcionario`. Por causa disso, não é possível chamar o método `qualSalário` a partir de uma instância da classe `Pessoa` - no primeiro método `calculaEmpréstimo`, que recebe um argumento que é uma instância da classe `Pessoa`, não podemos testar instâncias e chamar o método `qualSalário` se a instância for de `Funcionario` ou `ChefeDeDepartamento`, a não ser que efetuemos a conversão explícita (*cast*) das instâncias das classes.
  Assim como tipos nativos podem ser convertidos explicitamente para outros tipos seguindo certas regras, instâncias de classes também podem ser convertidas para outras, contanto que a conversão seja da classe mais específica para a mais genérica, ou seja, da classe descendente para a classe ancestral. Por exemplo, podemos fazer a conversão explícita de uma instância da classe `Funcionario` para a classe `Pessoa`, já que `Funcionario` *é-um-tipo-de* `Pessoa`. Similarmente, também podemos converter da classe `ChefeDeDepartamento` para a classe `Pessoa`.
  As regras de *cast* entre instâncias de classes não permitem a conversão de classes mais genéricas para classes mais específicas, nem de classes que não estejam em uma hierarquia de herança: não podemos converter da classe `Funcionario` para `ChefeDeDepartamento`, pois `Funcionario` não *é-um-tipo-de* `ChefeDeDepartamento`, nem da classe `Pessoa` para `String`. A listagem 8.16 demonstra o uso de conversão explícita entre as classes `Pessoa`, `Funcionario` e `ChefeDeDepartamento`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

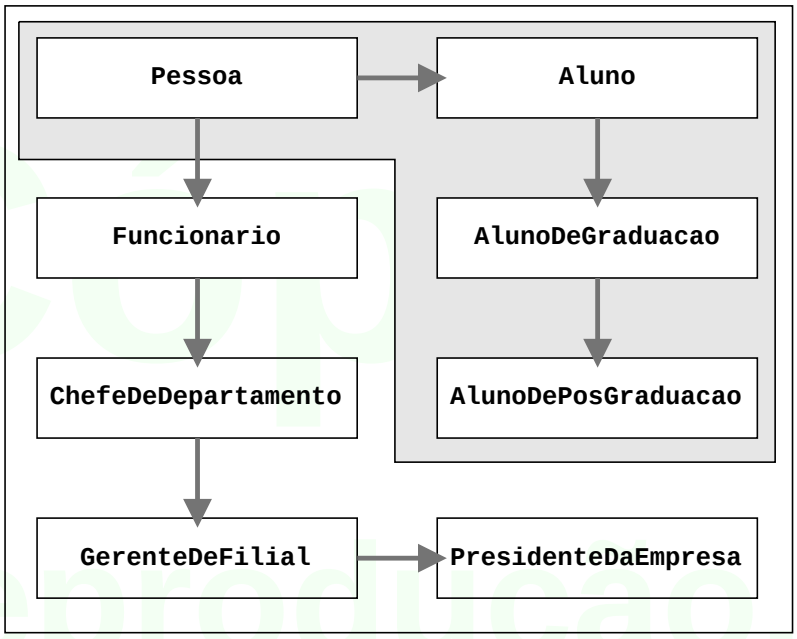



Figura 8.2: Herança com várias classes e separação para métodos polimórficos sobrecarregados

Listagem 8.16: A classe `EmprestimoBancarioComCast`, que demonstra polimorfismo e *cast* de instâncias de classes.

```
/**
 * A classe EmprestimoBancarioComCast, que calcula o valor de empréstimos que podem
 * ser dados a pessoas (instâncias das classes Pessoa, Funcionario e
 * ChefeDeDepartamento) por um banco popular. Esta classe é similar à classe
 * EmprestimoBancario, mas o método calculaEmpréstimo unifica o cálculo do empréstimo
 * que pode ser feito verificando a classe da instância passada como argumento e
 * fazendo o cast (conversão explícita) se necessário.
 */
class EmprestimoBancarioComCast
  {
  /**
   * O método main permite a execução desta classe. Este método contém declarações de
   * algumas instâncias das classes Pessoa, Funcionario e ChefeDeDepartamento, e as
   * usa para calcular o valor de um empréstimo bancário a ser feito para cada uma
   * das pessoas representadas pela instância.
   * @param argumentos os argumentos que podem ser passados para o método via linha
   *        de comando, mas que neste caso serão ignorados.
   */
  public static void main(String[] argumentos)
    {
    // Criamos uma instância da classe Pessoa
    Pessoa p1 = new Pessoa("Kurt Gödel",10973213,
                           new Data((byte)23,(byte)12,(short)1904));
    // Criamos três instâncias da classe Funcionario
    Funcionario f1 = new Funcionario("Henri Poincaré",19283712,
                                     new Data((byte)12,(byte)7,(short)1897),
                                     new Data((byte)28,(byte)1,(short)1918),
                                     2500.0f);
    Funcionario f2 = new Funcionario("Paul Dirac",98736812,
                                     new Data((byte)20,(byte)1,(short)1885),
                                     new Data((byte)31,(byte)3,(short)1909),
                                     3200.0f);
    Funcionario f3 = new Funcionario("Wolfgang Pauli",33886620,
                                     new Data((byte)14,(byte)9,(short)1902),
                                     new Data((byte)16,(byte)11,(short)1930),
                                     3600.0f);
    // Criamos uma instância da classe ChefeDeDepartamento
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
38      ChefeDeDepartamento c1 = new ChefeDeDepartamento("Edwin Hubble",4259782,
39                                                       new Data((byte)20,(byte)1,(short)1875),
40                                                       new Data((byte)20,(byte)7,(short)1899),
41                                                       4100.0f,
42                                                       "Laboratório de Astrofísica",
43                                                       new Data((byte)20,(byte)7,(short)1899));
44      // Calculamos os empréstimos que podem ser feitos a cada pessoa
45      System.out.println(calculaEmpréstimo(p1));
46      System.out.println(calculaEmpréstimo(f1));
47      System.out.println(calculaEmpréstimo(f2));
48      System.out.println(calculaEmpréstimo(f3));
49      System.out.println(calculaEmpréstimo(c1));
50      } // fim do método main
51
52    /**
53     * O método calculaEmpréstimo calcula o empréstimo que pode ser dado para uma
54     * pessoa (instância da classe Funcionario e suas descendentes). O empréstimo será
55     * calculado de forma diferenciada para as instâncias das classes Funcionario e
56     * ChefeDeDepartamento.
57     * @param f uma instância de qualquer classe que herde da classe Funcionario
58     */
59    public static float calculaEmpréstimo(Pessoa p)
60      {
61      float empréstimo = 1000.f; // inicialmente consideramos o empréstimo zero
62      // Primeiro verificamos se a instância p é uma instância da classe
63      // ChefeDeDepartamento. Se for, calculamos o empréstimo como sendo quatro vezes o
64      // salário recebido.
65      if (p instanceof ChefeDeDepartamento)
66        {
67        // Não podemos acessar o método qualSalário da instância p pois esta é uma
68        // instância da classe Pessoa. Devemos criar uma instância temporária da classe
69        // ChefeDeDepartamento a partir de p e usá-la para chamar o método qualSalário.
70        // Notem que não é necessário inicializar a instância com a palavra-chave new,
71        // ela será somente outra referência à p.
72        ChefeDeDepartamento temporário = (ChefeDeDepartamento)p;
73        empréstimo = 4.0f*temporário.qualSalário();
74        }
75      // Se a instância p não for da classe ChefeDeDepartamento, verificaremos se ela é
76      // instância da classe Funcionario, e se for, calculamos o empréstimo como sendo
77      // duas vezes o salário recebido.
78      else if (p instanceof Funcionario)
79        {
80        // Não podemos acessar o método qualSalário da instância p pois esta é uma
81        // instância da classe Pessoa. Devemos criar uma instância temporária da classe
82        // Funcionario a partir de p e usá-la para chamar o método qualSalário.
83        Funcionario temporário = (Funcionario)p;
84        empréstimo = 2.0f*temporário.qualSalário();
85        }
86      return empréstimo;
87      } // fim do método calculaEmpréstimo
88
89    } // fim da classe EmprestimoBancarioComCast
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

## 8.4 Exercícios do capítulo 8

**Exercício 8.1:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Ponto2D
2    {
3    private double x,y;
4    Ponto2D(double _x,double _y)
5      {
6      x = _x; y = _y;
7      }
8    }
9
10 class Ponto3D extends Ponto2D
11   {
12   private double z;
13   Ponto3D(double _x,double _y,double _z)
14     {
15     x = _x;
16     y = _y;
17     z = _z;
18     }
19   }
```

**Exercício 8.2:** ⋆

Explique com suas palavras porque podemos chamar implicitamente o método `toString` das instâncias da classe `Data` mas precisamos chamar explicitamente o mesmo método da classe ancestral dentro do método `toString` da classe `Funcionario` (listagem 8.8).

**Exercício 8.3:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DataHora extends Data,Hora
2    {
3    public DataHora(byte d,byte m,short a,byte hor,byte min,byte seg)
4      {
5      super(d,m,a);
6      super(hor,min,seg);
7      }
8    public String toString()
9      {
10     return super.toString()+" "+super.toString();
11     }
12   }
```

**Exercício 8.4:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Pai
2    {
3    private char x,y;
4    Pai(char _x,char _y)
5      {
6      x = _x; y = _y;
7      }
8    }
9
10 class Filho extends Pai
11   {
12   private char z;
13   public void inicializa(char _x,char _y,char _z)
14     {
15     super(_x,_y);
16     z = _z;
17     }
18   }
```

**Exercício 8.5:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DataHora extends Data
2    {
3    private Hora hora;
4    public DataHora(byte d,byte m,short a,byte hor,byte min,byte seg)
5      {
6      hora = new Hora(hor,min,seg);
7      super(d,m,a);
8      }
9    public String toString()
10     {
11     return super.toString()+" "+hora.toString();
12     }
13   }
```

**Exercício 8.6:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DataHora
2    {
3    private Data data;
4    private Hora hora;
5    public DataHora(byte d,byte m,short a,byte hor,byte min,byte seg)
6      {
7      super(d,m,a);
8      hora = new Hora(hor,min,seg);
9      }
10   public String toString()
11     {
12     return super.toString()+" "+hora.toString();
13     }
14   }
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 8.7:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DataHora extends Data,Hora
2    {
3    public DataHora(byte d,byte m,short a,byte hor,byte min,byte seg)
4      {
5      Data(d,m,a);
6      Hora(hor,min,seg);
7      }
8    public String toString()
9      {
10     return Data.toString()+" "+Hora.toString();
11     }
12   }
```

**Exercício 8.8:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  final class Ponto2D
2    {
3    private double x,y;
4    Ponto2D(double _x,double _y)
5      {
6      x = _x; y = _y;
7      }
8    }
9
10 final class Ponto3D extends Ponto2D
11   {
12   private double z;
13   Ponto3D(double _x,double _y,double _z)
14     {
15     super(_x,_y);
16     z = _z;
17     }
18   }
```

**Exercício 8.9:** ⋆

Explique com suas palavras o que aconteceria se removêssemos a palavra-chave `super` da linha 43 da classe `Funcionario` (listagem 8.8).

**Exercício 8.10:** ⋆

Liste todos os campos da classe `ChefeDeDepartamento` (listagem 8.9), inclusive os herdados, e indique quais podem ser modificados diretamente a partir da classe `ChefeDeDepartamento`.

**Exercício 8.11:** ⋆

Seria possível evitar completamente a necessidade de sobreposição de métodos criando métodos em classes descendentes que tenham assinaturas diferentes. Por exemplo, a classe `Pessoa` poderia ter o método `imprimePessoa` para imprimir seus campos, e a classe `Aluno` que estende a classe `Pessoa` poderia ter o método `imprimeAluno` para imprimir seus campos. Que vantagens e desvantagens esta abordagem teria sobre a sobreposição de métodos ?



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 8.12:** ⋆

Explique, com suas palavras, porque construtores de superclasses não são herdados por subclasses.

**Exercício 8.13:** ⋆

O que aconteceria se declarássemos o método `quantasPrestações` da classe `Automovel` (listagem 8.10) como sendo `final` ? Explique.

**Exercício 8.14:** ⋆

Para cada uma das cinco instâncias das classes criadas no método `main` da classe `EmprestimoBancario`, verifique se as expressões `instanceof Pessoa`, `instanceof Funcionario` e `instanceof ChefeDeDepartamento` retornariam verdadeiro ou falso.

**Exercício 8.15:** ⋆⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Pai
2    {
3    private char x,y;
4    Pai(char _x,char _y)
5      {
6      x = _x; y = _y;
7      }
8    }
9
10 class Filho extends Pai
11   {
12   private char z;
13   Filho(char _x,char _y,char _z)
14     {
15     z = _z;
16     super(_x,_y);
17     }
18   }
```

**Exercício 8.16:** ⋆⋆

Escreva a classe `LampadaFluorescente` como sendo herdeira da classe `Lampada`. A classe `LampadaFluorescente` deve ter um campo que represente o comprimento da lâmpada em centímetros. Crie nesta classe um construtor para inicializar os seus dados.

**Exercício 8.17:** ⋆⋆

Escreva, para a classe `Funcionario0` (listagem 8.5), um método `modificaNome` que permita a modificação do nome do funcionário. Considere as regras de modificação de acesso dos campos das classes envolvidas.

**Exercício 8.18:** ⋆⋆

Considere a classe `DataHora` (listagem 8.1) e as classes `Data` e `Hora` cujas instâncias são usadas na sua composição. Escreva, se ainda não existir na classe `Data`, um método `éIgual` que receba como argumento uma instância da própria classe `Data` e retorne o valor booleano `true` se a data representada for igual a data passada. Faça o mesmo para a classe `Hora`. Escreva também na classe `DataHora` um método `éIgual` que receba outra instância da própria classe `DataHora` como argumento e que seja executado delegando a comparação aos métodos das classes `Data` e `Hora`. Veja também a listagem 2.4.

**Exercício 8.19:** ⋆⋆

Escreva outro construtor para a classe `ChefeDeDepartamento` (listagem 8.9) que ao invés de receber e repassar os dados separados de um funcionário (`nome`, `identidade`, `admissão` e `nascimento`), receba uma instância da classe `Funcionario` já construída. Use o construtor da classe `DataHora` (listagem 8.1) como exemplo. Veja também o exercício 8.23.

**Exercício 8.20:** ⋆⋆

Escreva uma classe `Diretor` que herde da classe `ChefeDeDepartamento` e que contenha campos adicionais para representar a data de promoção ao cargo.

**Exercício 8.21:** ★★

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Data
2    {
3    private byte dia,mês; private short ano;
4    Data(byte d,byte m,short a)
5      {
6      dia = d; mês = m; ano = a;
7      }
8    public String toString()
9      {
10     return dia+"/"+mês+"/"+ano;
11     }
12   }
13
14 class DataHora extends Data
15   {
16   private Hora hora;
17   public DataHora(byte d,byte m,short a,byte hor,byte min,byte seg)
18     {
19     super(d,m,a);
20     hora = new Hora(hor,min,seg);
21     }
22   public String toString()
23     {
24     return super.toString()+" "+hora.toString();
25     }
26   }
27
28 class EntradaNaAgenda extends DataHora
29   {
30   private String evento;
31   public EntradaNaAgenda(byte d,byte m,short a,byte hor,byte min,byte seg,String ev)
32     {
33     super.super(d,m,a);
34     super(d,m,a,hor,min,seg);
35     evento = ev;
36     }
37   public String toString()
38     {
39     return super.super.toString()+":"+super.toString()+" -> "+ev;
40     }
41   }
```

**Exercício 8.22:** ★★

Escreva a classe `PacienteDeClinica`, descrita na figura 8.1.

**Exercício 8.23:** ★★

Escreva outro construtor para a classe `Funcionario` (listagem 8.8) que ao invés de receber e repassar os dados separados de uma pessoa (`nome` e `identidade`), receba uma instância da classe `Pessoa` já construída. Use o construtor da classe `DataHora` (listagem 8.1) como exemplo.

**Exercício 8.24:** ⋆⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Ponto2D
2    {
3    private double x,y;
4    Ponto2D(double _x,double _y)
5      {
6      x = _x; y = _y;
7      }
8    }
9
10 class Ponto3D extends Ponto2D
11   {
12   private double z;
13   Ponto3D(double _x,double _y,double _z)
14     {
15     z = _z;
16     }
17   }
```

**Exercício 8.25:** ⋆⋆

Escreva duas versões da classe `DataHora`: uma que usa o mecanismo de herança e que herda da classe `Data` e contém um campo que é uma instância da classe `Hora` e outra versão que herda da classe `Hora` e contém um campo que é uma instância da classe `Data`. Existem diferenças funcionais entre as classes ?

**Exercício 8.26:** ⋆⋆

Escreva as classes `LampadaTresEstados` e `Lampada100Estados`, que herdam da classe `Lampada`. Veja também os exercícios 2.28 e 2.29. Que métodos devem ser superpostos ?

**Exercício 8.27:** ⋆⋆

Reescreva as classes `Automovel` e `AutomovelDeLuxo` (listagens 8.10 e 8.12) para que o campo `NÚMEROMÁXIMODEPRESTAÇÕES` não seja mais usado, eliminando a necessidade de ocultá-lo. *Dica:* o valor do número máximo de prestações pode ser retornado como uma constante pelo método `quantasPrestações`, que pode ser superposto nas classes descendentes se o valor do número máximo de prestações for diferente das classes ancestrais. Alguma modificação deve ser feita na classe `AutomovelBasico` (listagem 8.11), que herda da classe `Automovel` e é a superclasse de `AutomovelDeLuxo` ? Alguma modificação deve ser feita na classe `DemoAutomoveis` (listagem 8.13), que demonstra os usos das classes ?

**Exercício 8.28:** ⋆⋆

Troque a ordem de verificação de instâncias na segunda versão do método `calculaEmpréstimo` da classe `EmprestimoBancario` (listagem 8.15). O que acontece ? Explique.

**Exercício 8.29:** ⋆⋆

Modifique os métodos `calculaEmpréstimo` da classe `EmprestimoBancario` (listagem 8.15) para que o empréstimo calculado para uma instância que herde da classe `Aluno` seja constante e igual a 500 reais.

**Exercício 8.30:** ⋆⋆

O que acontecerá se trocarmos a linha `Funcionario temporário = (Funcionario)p;` por `ChefeDeDepartamento temporário = (ChefeDeDepartamento)p;` dentro do `if` que verifica se a instância `p` é da classe `Funcionario` no método `calculaEmpréstimo` da classe `EmprestimoBancarioComCast` (listagem 8.16) ? Explique.

**Exercício 8.31:** ⋆⋆

O que acontecerá se o argumento para o método `calculaEmpréstimo` da classe `EmprestimoBancarioComCast` (listagem 8.16) for declarado como uma instância de `Object` ao invés de `Pessoa` ? Explique.

**Exercício 8.32:** ⋆⋆

Podemos fazer um *cast* indireto de classes para transformar uma instância da classe `AutomovelDeLuxo` em uma instância da classe `Object` e usar esta para criar uma instância da classe `String`, por exemplo ? Explique.

**Exercício 8.33:** ⋆⋆⋆

Escreva uma classe `EventoDelegacao` que seja baseada na classe `DataHora` e que contenha um campo para indicar qual o evento que ela representa (use uma String para isto). Use o mecanismo de delegação para criar a classe `EventoDelegacao`.

**Exercício 8.34:** ⋆⋆⋆

Escreva uma classe `EventoHeranca` que seja baseada na classe `DataHora` e que contenha um campo para indicar qual o evento que ela representa (use uma String para isto). Use o mecanismo de herança para criar a classe `EventoHeranca`. Veja também o exercício 8.33.

**Exercício 8.35:** ⋆⋆⋆

Usando o exercício 8.18 como base, crie na classe `Data` os métodos `éAntesDe` e `éDepoisDe` que retornam `true` se a data passada como argumento for respectivamente posterior e anterior à data representada. Escreva também estes métodos na classe `Hora`. Escreva também na classe `DataHora` os métodos `éAntesDe` e `éDepoisDe` que recebem uma instância da própria classe `DataHora` como argumento e que sejam executados delegando a comparação aos métodos das classes `Data` e `Hora`.

**Exercício 8.36:** ⋆⋆⋆

Escreva uma classe `VeiculoAVenda` que represente informações básicas sobre um veículo genérico que esteja à venda, como `tipo` (podendo ser representado por uma String), `ano` e `preçoDeVenda`. Escreva uma classe `AutomovelAVenda` que herde de `VeiculoAVenda` e que inicialize o campo `tipo` com o valor constante `"Automovel"`, e uma classe `MotocicletaAVenda` que herde de `VeiculoAVenda` e que inicialize o campo `tipo` com o valor constante `"Motocicleta"`. Escreva também um programa que demonstre o uso das diversas classes.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 8.37:** ⋆⋆⋆

Escreva as classes `LivroLivraria` e `LivroBiblioteca` que herdam da classe `Livro`. Quais as diferenças entre as duas classes, e que campos elas tem em comum ? *Dica:* os campos em comum devem ser preferencialmente representados pela classe ancestral. Veja também os exercícios 2.34, 2.35 e 2.36.

**Exercício 8.38:** ⋆⋆⋆

Escreva a classe `Politico` que herda da classe `Pessoa` e tem um campo adicional para representar o partido do político. Escreva também as classes `Prefeito` e `Governador` que herdem da classe `Politico` e que contenham campos para representar a cidade ou estado governado. Escreva também uma aplicação que demonstre o uso de instâncias destas classes.

**Exercício 8.39:** ⋆⋆⋆

Usando a classe `Ponto2D` (listagem 3.2) escreva duas versões da classe `Ponto3D`: uma usando o mecanismo de delegação e outra usando o mecanismo de herança.

**Exercício 8.40:** ⋆⋆⋆

Escreva a classe `ObjetoGeometrico` que represente um objeto geométrico em duas dimensões. Esta classe deve ter métodos para inicializar o objeto, mostrar seus dados e calcular e retornar a sua área e perímetro. Usando esta classe como base, escreva as classes herdeiras `Circulo` (contendo duas coordenadas para o centro e um raio), `Retangulo` (contendo dois valores para os lados) e `Triangulo` (contendo três valores para os lados), que sobrepõem os métodos descritos em `ObjetoGeometrico`. *Dicas:* a área de um círculo pode ser calculada com `Math.PI*r*r`, onde `r` é o raio do círculo. O perímetro de um círculo é dado por `2*Math.PI*r`. A área do retângulo é dada por `b*h` onde `b` é um dos lados e `h` é o outro lado. Seu perímetro é dado por `2*b+2*h`. A área de um triângulo é dada por `Math.sqrt(s*(s-a)*(s-b)*(s-c))` onde `Math.sqrt` é a função que calcula a raiz quadrada, `a`, `b` e `c` são os lados do triângulo e `s` é a metade do perímetro do triângulo. O perímetro do triângulo é calculado como `(a+b+c)`.

**Exercício 8.41:** ⋆⋆⋆

Usando o exercício 8.40 como base, escreva as classes `Quadrado`, que herda da classe `Retangulo` mas somente precisa inicializar um dos lados, e as classes `TrianguloEquilatero`, `TrianguloIsosceles` e `TrianguloEscaleno` que precisam inicializar somente um, dois ou três lados do triângulo. Para cada uma destas classes, quais métodos devem ser sobrepostos e quais podem ser aproveitados ?

## 8.5 Exercícios complementares do capítulo 8

**Exercício 8.42:** ⋆

Modifique o método `toString` da classe `RegistroAcademicoDeGraduacao` (listagem 8.2) para que o nome do curso seja impresso ao invés do seu código. Use, para isto, as constantes definidas na classe `RegistroAcademicoDeGraduacao`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 8.43:** ⋆

Escreva uma aplicação que use instâncias da classe `Pessoa` (listagem 8.7) e tente modificar diretamente os campos desta classe. O que acontece ?

**Exercício 8.44:** ⋆

Porque dentro do método `quantoCusta` da classe `Automovel` (listagem 8.10) a variável `preço` deve ser inicializada com zero ?

**Exercício 8.45:** ⋆

O valor 0 (zero) que é usado para inicializar a variável `preço` no método `quantoCusta` da classe `Automovel` (listagem 8.10). Explique, com suas palavras, por que esta constante não precisa ter o sufixo `f` (para explicitar o tipo `float`.

**Exercício 8.46:** ⋆

O método `imprime` da classe `ConcessionariaDeAutomoveis` (listagem 8.14) faz chamadas repetidas a um método da classe `Automovel`. Otimize o método `imprime` para minimizar o número de chamadas.

**Exercício 8.47:** ⋆⋆

Escreva uma aplicação em Java que declare e use algumas instâncias das classes `Pessoa`, `Funcionario` e `ChefeDeDepartamento`. Veja as classes nas listagens 8.7, 8.8 e 8.9 como referências.

**Exercício 8.48:** ⋆⋆

Os métodos `quantoCusta` e `toString` na classe `Automovel` (listagem 8.10) não prevêem o caso de alguma instância da classe ser criada com um valor para o combustível diferente dos previstos. Modifique os métodos para que estes possam tratar de outros valores não previstos, considerando que se o valor do campo `combustível` não for um dos quatro valores válidos, o combustível será considerado como sendo gasolina.

**Exercício 8.49:** ⋆⋆

Modifique a classe `AutomovelBasico` (listagem 8.11) para que esta tenha constantes finais booleanas `COMRETROVISOR`, `SEMRETROVISOR`, `COMLIMPADOR`, etc., de forma similar à classe `Automovel` e suas constantes para representar tipos de combustível. Faça o mesmo para a classe `AutomovelDeLuxo`, criando nesta as constantes `COMDIREÇÃOHIDRÁULICA`, etc.

**Exercício 8.50:** ⋆⋆⋆

Escreva um método `comQueIdadeEntrou` na classe `Funcionario` (listagem 8.8) que retorne a idade aproximada do funcionário quando ele foi admitido. Note que para isto talvez seja necessário escrever métodos na classe `Data` que permitam o acesso ao seu campo `ano`.

**Exercício 8.51:** ⋆⋆⋆

Modifique a classe `ConcessionariaDeAutomoveis` (listagem 8.14) para que um pequeno menu com os dados básicos (modelo e cor) de cada automóvel seja apresentado, permitindo ao usuário escolher um valor correspondente ao automóvel que o interessa. Quando o usuário entrar um valor, os dados completos do automóvel correspondente ao valor devem ser mostrados. *Dica:* vale a pena escrever um método `dadosBásicos` na classe `Automovel` que retorne, como uma String, somente o modelo e cor do automóvel.

# Capítulo 9

# Classes Abstratas e Interfaces

## 9.1 Introdução

No capítulo 8 vimos que o mecanismo de herança é uma poderosa ferramenta de orientação a objetos que permite a reutilização de código através da criação de classes baseadas em outras classes já existentes. Uma característica do mecanismo de herança é que deve existir uma superclasse a partir da qual as subclasses serão criadas, sendo que tanto a superclasse quanto as subclasses podem ser usadas para criar instâncias para uso em outras classes ou aplicações.

O planejamento de que classes terão herdeiras e do mecanismo de implementação das diferenças entre classes ancestrais e herdeiras (ocultação de campos, sobreposição de métodos) deve ser feito cuidadosamente, em especial para aplicações que envolvam muitas classes e vários níveis de herança. Em alguns casos, ao considerar que campos e métodos devem ser criados em uma superclasse, veremos que não é simples determinar quais campos e métodos deverão ser comuns a toda uma hierarquia de classes.

Com o mecanismo de herança visto até agora, devemos criar uma classe ancestral que tenha os campos e métodos comuns a todas as suas herdeiras, e devemos fazer a implementação dos métodos de forma que instâncias da classe ancestral possam ser criadas. Nem sempre isto é desejável - em alguns casos seria interessante descrever os campos e métodos que as classes herdeiras devem implementar, mas não permitir a criação de instâncias da classe ancestral. Desta forma a classe ancestral passaria a ser somente um guia de que métodos e campos deveriam ser implementados nas classes herdeiras. Em outras palavras, a classe ancestral ditaria para as classes descendentes *o que* deve ser feito, mas sem necessariamente dizer *como* deve ser feito.

Um exemplo pode ser dado pelas classes `Pessoa`, `Funcionario` e `ChefeDeDepartamento`, mostradas respectivamente nas listagens 8.7, 8.8 e 8.9. Para fins de representação em um organograma ou banco de dados sobre empregados de uma empresa, a classe `Pessoa` teria pouca utilidade, uma vez que somente instâncias da classe `Funcionario` ou herdeiras desta seriam utilizadas. Por outro lado, não faz sentido eliminarmos completamente a classe `Pessoa`, transferindo para `Funcionario` a tarefa de ser a superclasse, uma vez que outra aplicação (por exemplo, um controle acadêmico) poderia usar `Pessoa` como superclasse e `Aluno`, `AlunoDeGraduacao` e `AlunoDePosGraduacao` como descendentes, mas novamente, não haveria sentido em usar instâncias da classe `Pessoa` em uma aplicação de controle acadêmico. Idealmente, a classe `Pessoa` teria os campos e métodos em comum para as hierarquias de funcionários e alunos, mas seria tão genérica que não deveríamos poder criar instâncias dela.

Java tem dois mecanismos que permitem a criação de classes que somente contém descrições

de campos e métodos que devem ser implementados, mas sem efetivamente implementar estes métodos. Classes que declaram mas não implementam métodos são particularmente úteis na criação de hierarquias de classes como as descritas acima, porque não permitem a criação de instâncias delas e exigem que as classes descendentes implementem os métodos declarados nelas. Os dois mecanismos são *Classes abstratas* e *Interfaces*, e serão discutidos neste capítulo.

## 9.2 Classes abstratas

*Métodos abstratos*

O primeiro mecanismo de criação de superclasses com declarações mas sem definições de métodos permite a criação métodos declarados como *abstratos*. Métodos abstratos são somente declarados (com seu nome, modificadores, tipo de retorno e lista de argumentos), não tendo um corpo que contenha os comandos da linguagem que este método deva executar. Se uma classe contém um método declarado como abstrato, as classes que herdarem desta deverão obrigatoriamente implementar o método abstrato com o nome, modificador, tipo de retorno e argumentos declarados na classe ancestral[1].

`abstract`

*Classes abstratas*

Métodos abstratos são declarados com o modificador `abstract`. Se uma classe tiver algum método abstrato, a classe também deverá obrigatoriamente ser declarada com o modificador `abstract`. Uma classe também pode ser declarada abstrata mesmo que nenhum método tenha sido explicitamente declarado como abstrato. Uma classe que herde de uma classe abstrata deverá, obrigatoriamente, implementar todos os métodos declarados como abstratos na classe ancestral, se houverem métodos abstratos na classe ancestral. Caso não hajam, nenhuma implementação será obrigatória.

Como exemplo de classe abstrata vejamos o exemplo das classes que implementam robôs para simulação. Podemos considerar que existem mecanismos básicos que são comuns para todos os tipos de robôs, mas sua implementação será diferente dependendo do tipo de robô. Um bom exemplo é o mecanismo de movimento do robô: um robô simples pode se movimentar somente nas quatro direções cardeais, um robô mais complexo em várias direções angulares, um robô movido a energia limitada pode contabilizar quanta energia foi gasta no movimento, um robô com memória pode armazenar quantos passos foram dados, etc. - cada um destes mecanismos deveria ser implementado de maneira diferente, e não necessariamente tendo algo em comum - neste caso, implementar um método na classe ancestral e descrever as diferenças nas classes descendentes não seria muito simples ou útil. A classe ancestral poderia conter somente a declaração do método que executará o movimento, sem necessariamente conter os comandos para implementá-lo.

A listagem 9.1 mostra a classe abstrata `RoboAbstrato`, que contém campos básicos para representar um robô móvel simulado, mas declara um método como abstrato. Usaremos esta classe como superclasse para outras classes neste capítulo.

Listagem 9.1: A classe abstrata `RoboAbstrato`, que define que métodos e campos mínimos uma classe que implementa um robô deve conter.

```
1 /**
2  * A classe abstrata RoboAbstrato, que define que métodos e campos mínimos uma classe
3  * que implementa um robô deve conter.
4  * Esta classe é declarada como sendo abstrata, assim não pode ser instanciada
```

[1] É possível declarar um método em uma subclasse que sobrepõe um método abstrato em uma superclasse com modificadores diferentes, se o modificador não for mais restritivo na subclasse do que na superclasse, mas não existem aplicações simples e práticas desta capacidade.

```
5   * diretamente. Alguns de seus métodos tem corpo, podendo ser herdados diretamente
6   * por outra classe, ou sobrepostos. Os métodos que são declarados como sendo abstratos
7   * não tem corpo, e devem obrigatoriamente ser implementados pelas classes herdeiras
8   * (a não ser que as classes herdeiras sejam também abstratas).
9   */
10  abstract class RoboAbstrato // declaração da classe abstrata
11    {
12    /**
13     * Declaração dos campos da classe
14     */
15    private String nomeDoRobô;
16    private int posiçãoXAtual,posiçãoYAtual;
17    private short direçãoAtual; // permite maior flexibilidade na representação da
18                                // direção - consideraremos que são valores em graus
19
20    /**
21     * O construtor para a classe RoboAbstrato, que recebe argumentos para inicializar
22     * todos os campos da classe.
23     * @param n o nome do robô
24     * @param px a posição X atual
25     * @param py a posição Y atual
26     * @param d a direção atual
27     */
28    RoboAbstrato(String n,int px,int py,short d)
29      {
30      nomeDoRobô = n;
31      posiçãoXAtual = px;
32      posiçãoYAtual = py;
33      direçãoAtual = d;
34      } // fim do construtor com todos os argumentos
35
36    /**
37     * Esta versão do método move modifica a posição do robô em uma unidade na direção
38     * em que o robô está. Este método não recebe argumentos, e chama o método move com
39     * argumentos.
40     */
41    public void move()
42      {
43      move(1);
44      } // fim do método move
45
46    /**
47     * Esta versão do método move modifica a posição do robô em um número de unidades na
48     * direção em que o robô está. O método é declarado como abstrato e não tem corpo,
49     * pois não sabemos exatamente como um robô (instância de classe que herda desta) irá
50     * implementar seus movimentos.
51     * @param passos o número de "passos" para o robô
52     */
53    public abstract void move(int passos);
54
55    /**
56     * Este método permite a modificação do campo posiçãoXAtual, já que o método move
57     * foi declarado como abstrato e não existe outra maneira de modificar a posição
58     * do robô.
59     * @param passos o número de "passos" que serão usados para modificar a posição X
60     *        (horizontal) atual do robô
61     */
62    public void moveX(int passos)
63      {
64      posiçãoXAtual += passos;
65      }
66
67    /**
68     * Este método permite a modificação do campo posiçãoYAtual, já que o método move
69     * foi declarado como abstrato e não existe outra maneira de modificar a posição
70     * do robô.
71     * @param passos o número de "passos" que serão usados para modificar a posição Y
72     *        (vertical) atual do robô
73     */
```

```
74    public void moveY(int passos)
75      {
76      posiçãoYAtual += passos;
77      }
78
79    /**
80     * O método mudaDireção permite que a direção do robô seja mudada depois dele ter
81     * sido criado.
82     * @param novaDireção a nova direção para o robô.
83     */
84    public void mudaDireção(short novaDireção)
85      {
86      direçãoAtual = novaDireção;
87      } // fim do método mudaDireção
88
89    /**
90     * Este método permite a verificação do calor do campo direçãoAtual, já que o mesmo
91     * foi declarado como sendo privado.
92     * @return o valor do campo direçãoAtual
93     */
94    public short qualDireçãoAtual()
95      {
96      return direçãoAtual;
97      }
98
99    /**
100    * O método toString não recebe argumentos, e retorna uma String contendo os valores
101    * dos campos da classe formatados.
102    * @return uma String com os valores dos campos formatados.
103    */
104   public String toString()
105     {
106     String resultado = "Nome do robô:"+nomeDoRobô+"\n";
107     resultado = resultado+"Posição do robô: ("+posiçãoXAtual+","+
108                                               posiçãoYAtual+")\n";
109     resultado = resultado+"Direção do robô:"+direçãoAtual;
110     return resultado;
111     } // fim do método toString
112
113   } // fim da classe abstrata RoboAbstrato
```

*regras para criação de classes abstratas*

Alguns pontos interessantes da listagem 9.1 que exemplificam algumas das regras de criação de classes e métodos abstratos são:

- A classe `RoboAbstrato` deve ser declarada como abstrata, usando o modificador `abstract`, já que contém um método (`move`, sem argumentos) declarado como `abstract`.

*campos abstratos*

- Campos não podem ser declarados como `abstract`. Os campos de uma classe abstrata serão herdados pelas classes descendentes e poderão ser usados por instâncias destas a não ser que sejam declarados como `private`.
- A classe tem um construtor e vários métodos que não são declarados como abstratos. Estes métodos serão herdados pelas classes descendentes (já que não são privados), exceto pelo construtor, que não é tecnicamente herdado mas pode ser chamado pelos construtores de classes descendentes com a palavra-chave `super`.
- Mesmo se um método for declarado como abstrato, ele pode ser chamado a partir de outros, como no caso do método `move` sem argumentos, que chama o método `move` com argumentos, que por sua vez é abstrato. Isto não caracteriza um erro, porque espera-se que o método `move` sem argumentos só vá ser executado a partir de instâncias de classes que herdam da classe `RoboAbstrato`, e que obrigatoriamente deverão implementar o método `move` com argumentos..
- Métodos abstratos (como o método `move` sem argumentos) não podem ter corpo (a parte entre as chaves `{` e `}`). Somente a declaração do método é necessária, mas mesmo variáveis



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

passadas como argumento nunca sejam usadas, seus nomes devem ser especificados.

- Construtores de classes abstratas não podem ser abstratos. Mesmo que a classe abstrata não possa ser instanciada, seus construtores podem inicializar os campos da classe que serão usados por subclasses, sendo imprescindíveis em praticamente todos os casos.

Não podemos criar instâncias da classe `RoboAbstrato`, o que é intencional, uma vez que a classe `RoboAbstrato` somente declara os campos e métodos mínimos para um simulador de robôs, mas não implementando o método `move` (em outras palavras, dizendo que o método `move` deve ser implementando, mas não dizendo *como*). Podemos criar uma classe que herda da classe `RoboAbstrato` e que deverá obrigatoriamente implementar o método `move` - isto é demonstrado com a classe `RoboSimples`, mostrada na listagem 9.2.

Listagem 9.2: A classe `RoboSimples`, que herda da classe `RoboAbstrato` e que representa um robô de comportamento simples.

```
/**
 * A classe RoboSimples, que herda da classe RoboAbstrato e que representa um robô de
 * comportamento simples.
 * Esta classe herda todos os métodos não-abstratos da classe RoboAbstrato exceto o
 * construtor, que é redeclarado (mas aproveita o construtor da classe ancestral).
 * Esta classe também implementa o método move, que foi declarado como sendo abstrato
 * na classe ancestral. A classe não tem nenhum campo adicional.
 */
class RoboSimples extends RoboAbstrato // declaração da classe
  {
 /**
  * O construtor para a classe RoboSimples, que recebe argumentos para inicializar
  * todos os campos da classe, que foram herdados da classe RoboAbstrato. O construtor
  * simplesmente repassa os argumentos para o construtor da classe ancestral.
  * @param n o nome do robô
  * @param px a posição X atual
  * @param py a posição Y atual
  * @param d a direção atual
  */
  RoboSimples(String n,int px,int py,short d)
    {
    super(n,px,py,d);
    }

 /**
  * Esta versão do método move modifica a posição do robô em um número de unidades na
  * direção em que o robô está. O método foi declarado como sendo abstrato na classe
  * ancestral, e deve ser implementado nesta classe.
  * Consideramos que um robô simples pode se movimentar somente nas quatro direções
  * cardeais, ou seja, 0, 90, 180, 270 graus, correspondentes aos pontos cardeais
  * E, N, O, S. Outros valores são ignorados.
  * Como não podemos modificar diretamente os campos que determinam a posição atual do
  * robô, usamos os métodos moveX e moveY, herdados da classe ancestral.
  * @param passos o número de "passos" para o robô
  */
  public void move(int passos)
    {
    switch(qualDireçãoAtual()) // recuperamos a direção atual através deste método
      {
      case   0: moveX(+passos); break; // passos positivos na direção X
      case  90: moveY(+passos); break; // passos positivos na direção Y
      case 180: moveX(-passos); break; // passos negativos na direção X
      case 270: moveY(-passos); break; // passos negativos na direção Y
      }
    }

  } // fim da classe RoboSimples
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

A classe `RoboSimples`, na listagem 9.2, contém somente o construtor da classe e o método `move`, desta vez implementado. O construtor é obviamente necessário para a criação de instâncias da classe com inicialização de campos, e o método `move` teve que ser implementado por ser abstrato na classe ancestral. Na verdade podemos criar classes abstratas que herdam de classes abstratas: se algum método na classe herdeira `RoboSimples` for declarado como abstrato (ou mesmo se o método `move` não for implementado), a classe `RoboSimples` também deverá ser declarada como abstrata, seguindo as mesmas regras mostradas para a classe `RoboAbstrato`[2].

Como outro exemplo de classe que herda de uma classe abstrata (e tem que implementar seus métodos abstratos para que instâncias possam ser criadas), vejamos a classe `RoboABateria`, na listagem 9.3, que encapsula um robô simulado com mecanismo de movimento diferente do mostrado na classe `RoboSimples`.

Listagem 9.3: A classe `RoboABateria`, que estende a classe `RoboAbstrato` e representa um robô que consome energia com seus movimentos.

```
1  /**
2   * A classe RoboABateria, que herda da classe RoboAbstrato e que representa um robô
3   * que consome energia com seus movimentos.
4   * Esta classe herda todos os métodos não-abstratos da classe RoboAbstrato exceto o
5   * construtor. Esta classe também implementa o método move, que foi declarado como
6   * sendo abstrato na classe ancestral. A classe contém um campo adicional para indicar
7   * quantos passos o robô ainda pode dar.
8   */
9  class RoboABateria extends RoboAbstrato // declaração da classe
10   {
11   /**
12    * Declaração dos campos da classe
13    */
14    private long energia; // a energia que o robô ainda possui
15
16   /**
17    * O construtor para a classe RoboABateria, que recebe argumentos para inicializar
18    * todos os campos da classe. O construtor chama o construtor da classe ancestral
19    * repassando alguns argumentos, e inicializa um campo desta classe.
20    * @param n o nome do robô
21    * @param px a posição X atual
22    * @param py a posição Y atual
23    * @param d a direção atual
24    * @param e a energia inicial
25    */
26    RoboABateria(String n,int px,int py,short d,long e)
27      {
28      super(n,px,py,d);
29      energia = e;
30      }
31
32   /**
33    * Esta versão do método move modifica a posição do robô em um número de unidades na
34    * direção em que o robô está, e deduz o número de passos da energia que o robô tem,
35    * considerando que a cada passo dez unidades de energia são gastas.
36    * Consideramos que um robô a bateria pode se movimentar somente em oito direções,
37    * em intervalos de 45 graus, ou seja, nas direções 0, 45, 90, 135, 180, 225, 270 e
38    * 315 graus - outros valores são ignorados. Note que para os ângulos 45, 135, 225 e
39    * 315 graus o robô modificará as suas duas coordenadas. O método não faz nada caso
40    * a energia do robô não seja suficiente para movê-lo.
41    * @param passos o número de "passos" para o robô
42    */
43    public void move(int passos)
44      {
45      long energiaASerGasta = passos*10; // quanta energia será necessária para
```

[2] Podemos também criar classes abstratas que herdam de classes convencionais, mas esta é mais uma das características de Java que não tem muita aplicabilidade.

```
                                    // movimentar o robô ?
    if (energiaASerGasta <= energia) // o robô tem energia para se movimentar
      {
      switch(qualDireçãoAtual()) // recuperamos a direção atual através deste método
        {
        case   0: moveX(+passos); break; // passos positivos na direção X
        case  45: moveX(+passos);        // passos positivos na direção X e
                  moveY(+passos); break; // passos positivos na direção Y
        case  90: moveY(+passos); break; // passos positivos na direção Y
        case 135: moveY(+passos);        // passos positivos na direção Y
                  moveX(-passos); break; // passos negativos na direção X
        case 180: moveX(-passos); break; // passos negativos na direção X
        case 225: moveX(-passos);        // passos negativos na direção X e
                  moveY(-passos); break; // passos negativos na direção Y
        case 270: moveY(-passos); break; // passos negativos na direção Y
        case 315: moveY(-passos);        // passos negativos na direção Y e
                  moveX(+passos); break; // passos positivos na direção X
        } // fim do switch
      energia -= energiaASerGasta; // deduz a energia gasta para o movimento
      } // fim do if
    }

  /**
   * O método toString não recebe argumentos, e retorna uma String contendo os valores
   * dos campos da classe formatados. Este método chama o método toString da classe
   * ancestral e concatena ao resultado da chamada os campos exclusivos desta classe.
   * @return uma String com os valores dos campos formatados.
   */
  public String toString()
    {
    String resultado = super.toString()+"\n";
    resultado = resultado+"Energia do robô:"+energia;
    return resultado;
    } // fim do método toString

  } // fim da classe RoboABateria
```

A classe `DemoRobos`, mostrada na listagem 9.4 exemplifica o uso de instâncias das classes `RoboSimples` e `RoboABateria`. A classe não pode ser compilada por causa de um erro proposital (tentamos criar, dentro do método `main`, uma instância da classe abstrata `RoboAbstrato`).

Listagem 9.4: A classe `DemoRobos`, que demonstra instâncias e usos dos métodos das classes `RoboSimples` e `RoboABateria`.

```
/**
 * A classe DemoRobos, que demonstra instâncias das classes RoboSimples e
 * RoboABateria.
 * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
 */
class DemoRobos // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * instâncias das classes RoboSimples e RoboABateria, e tenta declarar uma instância
  * da classe abstrata RoboAbstrato (causando um erro de compilação).
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Criamos e manipulamos um robô que é uma instância da classe RoboSimples
    RoboSimples exp = new RoboSimples("Explorer",0,0,(short)90);
    exp.move(10);                // posição será 0,10
    exp.mudaDireção((short)180);
    exp.move();                  // posição será -1,10
    exp.move();                  // posição será -2,10
```

```
23      System.out.println(exp);
24      // Criamos e manipulamos um robô que é uma instância da classe RoboABateria
25      RoboABateria walk = new RoboABateria("Walker",0,0,(short)90,111);
26      walk.move(10);                  // posição será 0,10
27      walk.mudaDireção((short)180);
28      walk.move();                    // posição será -1,10
29      walk.move();                    // posição será -1,10 - não houve energia para
30                                      // modificar a posição
31      System.out.println(walk);
32      // Tentamos criar uma instância da classe RoboAbstrato, o que causará um erro de
33      // compilação.
34      RoboAbstrato imag = new RoboAbstrato("Imaginário",0,0,(short)180);
35      } // fim do método main
36
37  } // fim da classe DemoRobos
```

Curiosamente, classes abstratas podem ter métodos declarados como estáticos, contanto que estes não sejam abstratos. Métodos estáticos de classes abstratas podem ser executados da mesma forma que métodos estáticos comuns.

Métodos podem ter como tipo de retorno "instâncias" de classes abstratas, contanto que retornem classes não-abstratas que herdam da classe declarada como tipo de retorno do método. Por exemplo, a classe abstrata `RoboAbstrato` (listagem 9.1) poderia ter um método `public RoboAbstrato clona`, sem que isto caracterize um erro ou incoerência. Mesmo que seja impossível criar uma instância da classe `RoboAbstrato` para retorná-la como resultado do método `clona`, uma classe herdeira e não-abstrata poderia implementar o método e retornar uma instância de si mesma. Por exemplo, consideremos que a classe `RoboSaltador` herde da classe abstrata `RoboAbstrato` e implemente o método `public RoboAbstrato clona`, retornando uma nova instância de `RoboSaltador` como resultado do método. Desta forma o método seria válido, e estaria retornando uma instância válida de uma classe que *é-um-tipo-de* uma classe abstrata.

## 9.3 Interfaces

*interfaces*

Vimos que classes abstratas podem conter métodos não-abstratos que serão herdados e poderão ser utilizados por instâncias de classes herdeiras. Se a classe não tiver nenhum método não-abstrato, podemos criá-la como uma *interface*, que segue um modelo de declaração diferente do usado para classes mas tem funcionalidade similar a de classes abstratas.

Assim como uma classe abstrata, uma interface não pode ser instanciada. Todos os métodos na interface são implicitamente `abstract` e `public`, e não podem ser declarados com seus corpos. Campos, se houverem, serão implicitamente considerados como `static` e `final`, devendo, portanto, ser inicializados na sua declaração.

*comparação entre classes abstratas e interfaces*

A diferença essencial entre classes abstratas e interfaces é que uma classe herdeira somente pode herdar de uma única classe (abstrata ou não), enquanto que qualquer classe pode implementar várias interfaces simultaneamente. Interfaces são, então, um mecanismo simplificado de implementação de herança múltipla em Java, permitindo que mais de uma interface determine os métodos que uma classe herdeira deve implementar.

Interfaces também podem ser úteis para implementar bibliotecas de constantes: já que todos os campos de uma interface devem ser declarados como `static` e `final`, podemos escrever interfaces que somente contém campos, e qualquer classe que implementar esta interface terá acesso a estes campos.

Como primeiro exemplo do uso de interfaces, vejamos a interface `ObjetoGeometrico`, mostrada na listagem 9.5, que declara que métodos uma classe que represente um objeto geométrico genérico deve implementar.

Listagem 9.5: A interface `ObjetoGeometrico`, que representa um objeto geométrico bidimensional.

```
1  /**
2   * A interface ObjetoGeometrico, que define que métodos um objeto geométrico genérico
3   * bidimensional deve conter. Esta interface não declara nenhum campo.
4   */
5  interface ObjetoGeometrico
6    {
7   /**
8    * O método centro retorna o ponto em duas dimensões que corresponde ao centro
9    * do objeto geométrico.
10   * @return uma instância da classe Ponto2D representando o centro do objeto
11   *         geométrico.
12   */
13   Ponto2D centro();
14
15  /**
16   * O método calculaÁrea calcula a área do objeto geométrico e retorna a área
17   * como um valor do tipo double.
18   * @return a área deste objeto geométrico.
19   */
20   double calculaÁrea();
21
22  /**
23   * O método calculaPerímetro calcula o perímetro do objeto geométrico e retorna o
24   * perímetro como um valor do tipo double.
25   * @return o perímetro deste objeto geométrico.
26   */
27   double calculaPerímetro();
28
29   } // fim da interface ObjetoGeometrico
```

Alguns pontos interessantes da listagem 9.5 são:

- Interfaces são declaradas com a palavra-chave `interface` e não com a palavra-chave `class`.
- A declaração dos métodos em uma interface é feita como a declaração dos métodos em uma classe abstrata, mas sem o modificador `abstract`. Nenhum método em uma interface pode ter corpo.
- Os métodos em uma interface são, implicitamente, `abstract` e `public`, não precisando ser declarados desta forma. Métodos em interfaces não podem ter modificadores como `static`, `private`, etc.
- Interfaces não podem ter construtores, mesmo declarados como métodos abstratos e públicos.
- Assim como uma classe comum, interfaces podem utilizar outras classes como argumentos ou tipos de retorno para seus métodos, como mostrado pelo método `centro`, que deverá retornar uma instância da classe `Ponto2D` quando for implementado.

Para exemplificar o uso da interface `ObjetoGeometrico`, vamos criar algumas classes não-abstratas que implementam a interface. O primeiro exemplo é mostrado na classe `Circulo`, na listagem 9.6.

Listagem 9.6: A classe `Circulo`, que implementa a interface `ObjetoGeometrico` e representa um círculo.

```
1  /**
2   * A classe Circulo, que implementa a interface ObjetoGeometrico e representa um
3   * círculo. Esta classe encapsula o ponto central do círculo e seu raio. Todos os
4   * métodos declarados na interface ObjetoGeometrico são implementados.
5   */
6  class Circulo implements ObjetoGeometrico // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10   */
11   private Ponto2D centro;
12   private double raio;
13
14  /**
15   * O construtor para a classe Circulo, que recebe argumentos para inicializar todos
16   * os campos da classe. Este é o construtor "completo".
17   * @param centro o centro do círculo (uma instância da classe Ponto2D)
18   * @param raio o raio do círculo
19   */
20   Circulo(Ponto2D centro,double raio)
21     {
22     this.centro = centro; // Note o uso da palavra-chave this para marcar os campos
23     this.raio = raio;     // da classe que tem o mesmo nome dos argumentos
24     }
25
26  /**
27   * O método centro retorna o ponto em duas dimensões que corresponde ao centro do
28   * círculo. Neste caso, basta retornar a instância da classe Ponto2D que representa
29   * o próprio centro.
30   * @return uma instância da classe Ponto2D representando o centro do círculo
31   */
32   public Ponto2D centro()
33     {
34     return centro;
35     }
36
37  /**
38   * O método calculaÁrea calcula a área do círculo e retorna a área como um valor do
39   * tipo double.
40   * @return a área deste círculo
41   */
42   public double calculaÁrea()
43     {
44     return Math.PI*raio*raio; // pi vezes o quadrado do raio
45     }
46
47  /**
48   * O método calculaPerímetro calcula o perímetro do círculo e retorna o perímetro
49   * como um valor do tipo double.
50   * @return o perímetro deste círculo
51   */
52   public double calculaPerímetro()
53     {
54     return 2.0*Math.PI*raio; // dois vezes pi vezes o raio
55     }
56
57  /**
58   * O método toString retorna os valores dos campos do círculo formatados em uma
59   * String.
60   * @return uma String contendo uma representação dos campos do círculo
61   */
62   public String toString()
63     {
64     // Retorna diretamente uma String criada com os valores dos campos. O método
65     // toString da classe Ponto2D é chamado implicitamente.
66     return "Círculo com centro em "+centro+" e raio "+raio;
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
67      }
68
69    } // fim da classe Circulo
```

Algumas observações sobre a classe `Circulo` na listagem 9.6 são:

- A relação de herança entre uma interface e uma classe é feita declarando-se a classe como implementando a interface, usando-se a palavra-chave `implements`, na forma `class AClasse implements AInterface`.
- A classe `Circulo` deve implementar todos os métodos que foram declarados na interface `ObjetoGeometrico`. Se algum método não for declarado, a classe que implementa a interface deverá ser declarada como abstrata.
- O método `centro`, declarado na interface `ObjetoGeometrico` sem modificadores, apresenta o modificador `public` na sua implementação, o que não caracteriza erros, pois o método era implicitamente `public` na interface. O método não poderia ser declarado como `private` ou `protected` (ou mesmo sem modificador !) pois não é possível fazer a sobrecarga de um método se o acesso é tornado mais restritivo. O método também não poderia ser declarado como estático.

Outro exemplo de uso da interface `ObjetoGeometrico` é dado pela classe `Retangulo`, mostrada na listagem 9.7.

Listagem 9.7: A classe `Retangulo`, que implementa a interface `ObjetoGeometrico` e representa um retângulo.

```
1  /**
2   * A classe Retangulo, que implementa a interface ObjetoGeometrico e representa um
3   * retângulo. Esta classe encapsula os dois pontos extremos do retângulo. Todos os
4   * métodos declarados na interface ObjetoGeometrico são implementados.
5   */
6  class Retangulo implements ObjetoGeometrico // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10   */
11   private Ponto2D primeiroCanto,segundoCanto;
12
13  /**
14   * O construtor para a classe Retangulo, que recebe argumentos para inicializar todos
15   * os campos da classe. Este é o construtor "completo".
16   * @param pc o primeiro canto do retângulo (uma instância da classe Ponto2D)
17   * @param sc o segundo canto do retângulo (outra instância da classe Ponto2D)
18   */
19   Retangulo(Ponto2D pc,Ponto2D sc)
20     {
21     primeiroCanto = pc;
22     segundoCanto = sc;
23     }
24
25  /**
26   * O método centro retorna o ponto em duas dimensões que corresponde ao centro do
27   * retângulo. Neste caso, basta calcular o ponto equidistante dos dois pontos que
28   * delimitam o retângulo, que é calculado usando a média das coordenadas X e Y dos
29   * cantos.
30   * @return uma instância da classe Ponto2D representando o centro do retângulo
31   */
32   public Ponto2D centro()
33     {
34     double coordX = (primeiroCanto.getX()+segundoCanto.getX())/2.;
35     double coordY = (primeiroCanto.getY()+segundoCanto.getY())/2.;
36     return new Ponto2D(coordX,coordY);
```

```
37      }
38
39    /**
40     * O método calculaÁrea calcula a área do retângulo e retorna a área como um valor do
41     * tipo double.
42     * @return a área deste retângulo
43     */
44    public double calculaÁrea()
45      {
46      // Calcula o lado X do retângulo, que é a diferença entre as coordenadas X do
47      // do primeiro e segundo canto do retângulo. O método Math.abs garante que o valor
48      // será positivo.
49      double ladoX = Math.abs(primeiroCanto.getX()-segundoCanto.getX());
50      // Calcula o lado Y do retângulo (diferença entre as coordenadas Y dos cantos)
51      double ladoY = Math.abs(primeiroCanto.getY()-segundoCanto.getY());
52      return ladoX*ladoY; // multiplicação dos comprimentos dos lados
53      }
54
55    /**
56     * O método calculaPerímetro calcula o perímetro do retângulo e retorna o perímetro
57     * como um valor do tipo double.
58     * @return o perímetro deste retângulo
59     */
60    public double calculaPerímetro()
61      {
62      // Calcula o lado X do retângulo, que é a diferença entre as coordenadas X do
63      // do primeiro e segundo canto do retângulo. O método Math.abs garante que o valor
64      // será positivo.
65      double ladoX = Math.abs(primeiroCanto.getX()-segundoCanto.getX());
66      // Calcula o lado Y do retângulo (diferença entre as coordenadas Y dos cantos)
67      double ladoY = Math.abs(primeiroCanto.getY()-segundoCanto.getY());
68      return 2*ladoX+2*ladoY; // soma do dobro dos lados
69      }
70
71    /**
72     * O método toString retorna os valores dos campos do retângulo formatados em uma
73     * String.
74     * @return uma String contendo uma representação dos campos do retângulo
75     */
76    public String toString()
77      {
78      // Retorna diretamente uma String criada com os valores dos campos. O método
79      // toString da classe Ponto2D é chamado implicitamente.
80      return "Retângulo com cantos "+primeiroCanto+" e "+segundoCanto;
81      }
82
83    } // fim da classe Retangulo
```

As classes `Circulo` e `Retangulo` mostram como os métodos da interface `ObjetoGeometrico` devem ser implementados de maneira diferente. O centro de um `Circulo` é armazenado diretamente na classe, enquanto que o centro de um `Retangulo` deve ser calculado como sendo o ponto médio de suas extremidades. Similarmente, a maneira de calcular a área e o perímetro de círculos e retângulos é diferente, sendo calculada de maneira diferente nas duas classes.

A classe `DemoObjetosGeometricos`, na listagem 9.8, demonstra usos de instâncias das classes `Circulo` e `Retangulo`.

Listagem 9.8: A classe `DemoObjetosGeometricos`, que demonstra instâncias de classes que implementam a interface `ObjetoGeometrico`.

```
1   /**
2    * A classe DemoObjetosGeometricos, que demonstra instâncias de classes que
3    * implementam a interface ObjetoGeometrico.
4    */
```

```
5  class DemoObjetosGeometricos // declaração da classe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * instâncias das classes Circulo e Retangulo, duas classes que implementam a
10   * interface ObjetoGeometrico.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Declaramos algumas instâncias das classes Circulo e Retangulo
17     Circulo c1 = new Circulo(new Ponto2D(0,0),100); // centrado na origem, raio 100
18     Circulo c2 = new Circulo(new Ponto2D(-1,-1),1); // centrado em (-1,-1), raio 1
19     Circulo c3 = new Circulo(new Ponto2D(10,8),0); // centrado em (10,8), raio 0 (!)
20     Retangulo r1 = new Retangulo(new Ponto2D(-2,-2), // um canto está em (-2,-2)
21                                  new Ponto2D(2,2));  // e o outro em (2,2)
22     Retangulo r2 = new Retangulo(new Ponto2D(-100,-1), // um canto está em (-100,1)
23                                  new Ponto2D(100,1));  // e o outro em (100,1)
24     Retangulo r3 = new Retangulo(new Ponto2D(0,0),  // um canto está em (0,0)
25                                  new Ponto2D(0,0)); // e o outro também !
26     // Vamos imprimir os dados completos de cada um destes objetos geométricos
27     imprimeTodosOsDados(c1);
28     imprimeTodosOsDados(c2);
29     imprimeTodosOsDados(c3);
30     imprimeTodosOsDados(r1);
31     imprimeTodosOsDados(r2);
32     imprimeTodosOsDados(r3);
33     } // fim do método main
34
35   /**
36    * O método imprimeTodosOsDados imprime todos os dados de uma instância de uma
37    * classe que implemente a interface ObjetoGeometrico.
38    * @param og uma instância de qualquer classe que implemente a interface
39    *           ObjetoGeometrico.
40    */
41   private static void imprimeTodosOsDados(ObjetoGeometrico og)
42     {
43     // Primeiro, imprime os dados padrão do objeto - podemos fazer isto diretamente
44     // através da chamada implícita ao método toString
45     System.out.println(og);
46     // Podemos imprimir a sua área e perímetro com chamadas diretas aos métodos das
47     // classes
48     System.out.println("Perímetro:"+og.calculaPerímetro());
49     System.out.println("Área:"+og.calculaÁrea());
50     // Imprimimos uma linha em branco para melhor visualização dos resultados
51     System.out.println();
52     } // fim do método imprimeTodosOsDados
53
54   } // fim da classe DemoObjetosGeometricos
```

*interfaces e polimorfismo*

A classe `DemoObjetosGeometricos`, na listagem 9.8, demonstra um importante conceito envolvendo interfaces: se uma classe implementa uma interface, uma instância desta classe pode ser passada como argumento para qualquer método que espere como argumento uma referência à interface implementada. Este é o mesmo mecanismo de polimorfismo apresentado na seção 8.3.3 - embora não possamos criar instâncias de interfaces, a referência `og` no método `imprimeTodosOsDados` na classe `DemoObjetosGeometricos` pode comportar-se como uma instância de `Circulo` ou `Retangulo`, então podemos chamar os métodos que foram declarados na interface.

Um exemplo mais detalhado de polimorfismo em classes que implementam uma interface em comum é mostrado na classe `DemoObjetosGeometricosEPolimorfismo` (listagem 9.9).

Listagem 9.9: A classe `DemoObjetosGeometricosEPolimorfismo`, que demonstra características de polimorfismo com interfaces e classes que as implementam.

```
/**
 * A classe DemoObjetosGeometricosEPolimorfismo, que demonstra características de
 * polimorfismo com interfaces e classes que as implementam.
 */
class DemoObjetosGeometricosEPolimorfismo // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * referências da interface ObjetoGeometrico, que receberão instâncias das classes
  * Circulo e Retangulo, classes que implementam a interface.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Declaramos algumas referências à interface ObjetoGeometrico
    ObjetoGeometrico o1,o2,o3,o4;
    // Criamos as instâncias das classes Circulo e Retangulo e as associamos às
    // referências à interface ObjetoGeometrico. Note que o cast não é necessário,
    // já que um Circulo (ou Retangulo) é um tipo de ObjetoGeometrico
    o1 = new Circulo(new Ponto2D(0,0),20);     // centrado na origem, raio 20
    o2 = new Retangulo(new Ponto2D(-1,-1),     // um canto está em (-1,-1)
                       new Ponto2D(1,1));      // e o outro em (1,1)
    o3 = new Circulo(new Ponto2D(-7,3),12.3); // centrado em (-7,3), raio 12.3
    o4 = new Retangulo(new Ponto2D(0,0),       // um canto está em (0,0)
                       new Ponto2D(1,1));      // e o outro em (1,1)
    // Vamos ver que referência é instância de que classe
    System.out.println("o1 é um Círculo ? "+
                       (o1 instanceof Circulo)); // true
    System.out.println("o1 é um Retângulo ? "+
                       (o1 instanceof Retangulo)); // false
    System.out.println("o1 é um ObjetoGeometrico ? "+
                       (o1 instanceof ObjetoGeometrico)); // true
    System.out.println("o2 é um Círculo ? "+
                       (o2 instanceof Circulo)); // false
    System.out.println("o2 é um Retângulo ? "+
                       (o2 instanceof Retangulo)); // true
    System.out.println("o2 é um ObjetoGeometrico ? "+
                       (o2 instanceof ObjetoGeometrico)); // true
    } // fim do método main

  } // fim da classe DemoObjetosGeometricosEPolimorfismo
```

No método `main` da classe `DemoObjetosGeometricosEPolimorfismo` declaramos várias referências da interface `ObjetoGeometrico` - embora seja impossível criar instâncias desta interface podemos declarar referências a ela, que poderão apontar para instâncias de classes que implementam a interface.

Após a criação destas instâncias, verificamos a que classe cada instância pertence, com a palavra-chave `instanceof`. Com o resultado do programa podemos ver que qualquer uma das instâncias pode ser considerada como sendo da sua própria classe e da interface `ObjetoGeometrico` - em outras palavras, as instâncias *são-um-tipo-de* `ObjetoGeometrico`. É importante notar que classes que herdam de uma interface comum não são necessariamente consideradas como sendo similares - de acordo com a classe `DemoObjetosGeometricosEPolimorfismo`, instâncias da classe `Circulo` não são consideradas como instâncias da classe `Retangulo` e vice-versa.

`instanceof`

## 9.4 Herança múltipla usando interfaces

*herança múltipla*

Conforme comentado anteriormente, a principal diferença entre herança usando classes abstratas e usando interfaces é que uma classe pode herdar somente de uma única classe, enquanto pode implementar diversas interfaces.

Um exemplo deste mecanismo é mostrado a seguir. Consideremos que alguns objetos geométricos podem ser escaláveis, isto é, seu tamanho original pode ser modificado usando-se um valor como escala. Os dados que representam o tamanho do objeto seriam modificados por um método que recebesse a escala como argumento.

A interface `Escalavel`, mostrada na listagem 9.10, declara os métodos que um objeto geométrico escalável deve implementar. Além do método que permite a modificação do tamanho baseado em uma escala, a interface também declara um método que permite o espelhamento do objeto, isto é, a modificação de sua posição horizontal.

Listagem 9.10: A interface `Escalavel`, que define que métodos um objeto que seja escalável (possa ter seu tamanho modificável).

```
1  /**
2   * A interface Escalavel, que define que métodos um objeto que seja escalável (isto é,
3   * que pode ter seu tamanho modificado como uma função do tamanho anterior) deve
4   * conter. Esta interface não declara nenhum campo.
5   */
6  interface Escalavel
7    {
8   /**
9    * O método amplia modifica os campos do objeto para alterar o seu tamanho.
10   * @param escala a escala para modificação do objeto
11   */
12   void amplia(double escala);
13
14  /**
15   * O método espelha modifica os campos do objeto para alterar a sua posição (fazendo
16   * com que o objeto fique refletido nas suas coordenadas horizontais)
17   */
18   void espelha();
19
20   } // fim da interface Escalavel
```

A interface `Escalavel` não contém campos, e somente declara os dois métodos que devem ser definidos em classes que implementam esta interface.

A classe `CirculoEscalavel`, mostrada na listagem 9.11, implementa duas interfaces simultaneamente (`ObjetoGeometrico` e `Escalavel`). Esta classe deve conter os métodos declarados nas duas interfaces, e demonstra o mecanismo de herança múltipla de Java.

Listagem 9.11: A classe `CirculoEscalavel`, que implementa as interfaces `ObjetoGeometrico` e `Escalavel`.

```
1  /**
2   * A classe Circulo, que implementa as interfaces ObjetoGeometrico e Escalavel, e
3   * representa um círculo cujo tamanho pode ser modificado em função do seu tamanho
4   * anterior. Esta classe encapsula o ponto central do círculo e seu raio. Todos os
5   * métodos declarados nas interfaces ObjetoGeometrico e Escalavel são implementados.
6   */
7  class CirculoEscalavel implements ObjetoGeometrico,Escalavel // declaração da classe
8    {
9   /**
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
10    * Declaração dos campos da classe
11    */
12   private Ponto2D centro;
13   private double raio;
14
15  /**
16   * O construtor para a classe CirculoEscalavel, que recebe argumentos para
17   * inicializar todos os campos da classe. Este é o construtor "completo".
18   * @param centro o centro do círculo (uma instância da classe Ponto2D)
19   * @param raio o raio do círculo
20   */
21   CirculoEscalavel(Ponto2D centro,double raio)
22     {
23     this.centro = centro; // Note o uso da palavra-chave this para marcar os campos
24     this.raio = raio;     // da classe que tem o mesmo nome dos argumentos
25     }
26
27  /**
28   * O método centro retorna o ponto em duas dimensões que corresponde ao centro do
29   * círculo. Neste caso, basta retornar a instância da classe Ponto2D que representa
30   * o próprio centro.
31   * @return uma instância da classe Ponto2D representando o centro do círculo
32   */
33   public Ponto2D centro()
34     {
35     return centro;
36     }
37
38  /**
39   * O método calculaÁrea calcula a área do círculo e retorna a área como um valor do
40   * tipo double.
41   * @return a área deste círculo
42   */
43   public double calculaÁrea()
44     {
45     return Math.PI*raio*raio; // pi vezes o quadrado do raio
46     }
47
48  /**
49   * O método calculaPerímetro calcula o perímetro do círculo e retorna o perímetro
50   * como um valor do tipo double.
51   * @return o perímetro deste círculo
52   */
53   public double calculaPerímetro()
54     {
55     return 2.0*Math.PI*raio; // dois vezes pi vezes o raio
56     }
57
58  /**
59   * O método amplia modifica o raio do círculo de acordo com o valor passado como
60   * argumento, efetivamente modificando o tamanho do círculo.
61   * @param escala a escala para modificação do círculo
62   */
63   public void amplia(double escala)
64     {
65     raio *= escala;
66     }
67
68  /**
69   * O método espelha modifica o centro do círculo para alterar a sua posição (fazendo
70   * com que o círculo fique refletido nas suas coordenadas horizontais)
71   */
72   public void espelha()
73     {
74     centro = new Ponto2D(-centro.getX(),centro.getY());
75     }
76
77  /**
78   * O método toString retorna os valores dos campos do círculo formatados em uma
```

```
79    * String.
80    * @return uma String contendo uma representação dos campos do círculo
81    */
82    public String toString()
83      {
84      // Retorna diretamente uma String criada com os valores dos campos. O método
85      // toString da classe Ponto2D é chamado implicitamente.
86      return "Círculo com centro em "+centro+" e raio "+raio;
87      }
88
89    } // fim da classe CirculoEscalavel
```

Um ponto interessante da classe `CirculoEscalavel` é a implementação do método `espelha`, cuja declaração foi herdada da interface `Escalavel` - o método simplesmente cria outra instância da classe `Ponto2D` que representa o centro do círculo, de forma que a coordenada horizontal da nova instância é o valor negativo da coordenada horizontal do valor anterior. A classe `DemoCirculoEscalavel` (mostrada na listagem 9.12) demonstra os métodos da classe `CirculoEscalavel` e a relação entre instâncias desta classe e as interfaces que ela implementa.

Listagem 9.12: A classe `DemoCirculoEscalavel`, que demonstra o uso dos métodos da classe `CirculoEscalavel`.

```
1  /**
2   * A classe DemoCirculoEscalavel, que demonstra o uso dos métodos da classe
3   * CirculoEscalavel e sua relação com as interfaces ObjetoGeometrico e Escalavel.
4   */
5  class DemoCirculoEscalavel // declaração da classe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * uma instância da classe CirculoEscalavel e usa seus métodos.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     // Declaramos e alocamos uma instância da classe CirculoEscalavel
16     CirculoEscalavel ce = new CirculoEscalavel(new Ponto2D(10,10),30);
17     // Imprimimos a posição original do círculo escalável
18     System.out.println(ce); // chamada implícita ao método toString
19     // Modificamos a escala do círculo
20     ce.amplia(3);
21     // Imprimimos novamente os dados do círculo
22     System.out.println(ce);
23     // Modificamos a posição do círculo (espelhando)
24     ce.espelha();
25     // Imprimimos novamente os dados do círculo
26     System.out.println(ce);
27     // É uma instância da classe CirculoEscalavel ?
28     System.out.println(ce instanceof CirculoEscalavel); // true
29     // É uma instância da interface ObjetoGeometrico ?
30     System.out.println(ce instanceof ObjetoGeometrico); // true
31     // É uma instância da interface Escalavel ?
32     System.out.println(ce instanceof Escalavel);        // true
33     } // fim do método main
34
35   } // fim da classe DemoCirculoEscalavel
```

*herança múltipla com classes e interfaces*

Também é possível implementar herança múltipla em Java fazendo com que uma classe herde de outra mas implemente uma ou mais interfaces. As regras de herança múltipla com esta abordagem não são diferentes das regras de herança de classes e implementação de interfaces: uma classe herda todos os métodos e campos que não forem privados da classe ancestral e deve implementar todos os métodos declarados nas interfaces.

Um exemplo de herança múltipla de classes e interfaces é dada a seguir. Primeiro, criamos uma interface `ItemDeBiblioteca` que declara que campos e métodos uma classe que representa um item para empréstimo em uma biblioteca deve implementar. A interface `ItemDeBiblioteca` é mostrada na listagem 9.13.

Listagem 9.13: A interface `ItemDeBiblioteca`, que declara os campos e métodos que qualquer item em uma biblioteca deve ter.

```
/**
 * A interface ItemDeBiblioteca, que declara os campos e métodos que qualquer item
 * em uma biblioteca deve ter.
 */
interface ItemDeBiblioteca
  {
 /**
  * Declaração dos campos da interface - note que todo campo em uma interface é
  * considerado como public final, não podendo variar.
  */
  int máximoDeDiasParaEmpréstimo = 14;

 /**
  * O método estáEmprestado retorna true se o item estiver emprestado e false caso
  * contrário.
  * @return true se o item estiver emprestado e false caso contrário
  */
  boolean estáEmprestado();

 /**
  * O método empresta modifica o estado de um campo que indica se o item está
  * emprestado ou não para true.
  */
  void empresta();

 /**
  * O método devolve modifica o estado de um campo que indica se o item está
  * emprestado ou não para false.
  */
  void devolve();

 /**
  * O método localização retorna a localização do item na biblioteca
  * @return a localização do item na biblioteca
  */
  String localização();

 /**
  * O método descriçãoSumária retorna uma String contendo uma descrição sumária do
  * item de biblioteca
  * @return a descrição sumária do item da biblioteca
  */
  String descriçãoSumária();

  } // fim da interface ItemDeBiblioteca
```

A interface `ItemDeBiblioteca` tem um campo, `máximoDeDiasParaEmpréstimo`, que é implicitamente `static` e `final`, e algumas declarações de métodos que devem ser criados nas classes que implementam esta interface.

Outra classe que será usada no exemplo de herança múltipla é a classe `Livro`, que encapsula os dados genéricos sobre um livro e métodos para processar estes dados. A classe `Livro` é mostrada na listagem 9.14.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem 9.14: A classe `Livro`, que encapsula os dados de um livro.

```
1  /**
2   * A classe Livro, que encapsula os dados de um livro. Para simplificar, somente os
3   * campos e métodos mais básicos são implementados nesta classe.
4   */
5  class Livro
6    {
7   /**
8    * Declaração dos campos da classe
9    */
10   private String título,autor; // o título e autor do livro
11   private short númeroDePáginas; // o número de páginas do livro
12   private short anoDaEdição; // o ano da edição do livro
13
14  /**
15   * O construtor para a classe Livro, que recebe argumentos para inicializar todos
16   * os campos da classe. Este é o construtor "completo".
17   * @param tit o título do livro
18   * @param aut o(s) nome(s) do(s) autor(es) do livro
19   * @param np o número de páginas do livro
20   * @param ae o ano da edição do livro
21   */
22   Livro(String tit,String aut,short np,short ae)
23     {
24     título = tit; autor = aut; númeroDePáginas = np; anoDaEdição = ae;
25     }
26
27  /**
28   * O método qualTítulo retorna o título deste livro.
29   * @return o título do livro
30   */
31  public String qualTítulo()
32    {
33    return título;
34    }
35
36  /**
37   * O método qualAutor retorna o autor deste livro.
38   * @return o autor do livro
39   */
40  public String qualAutor()
41    {
42    return autor;
43    }
44
45  /**
46   * O método toString retorna os valores dos campos desta classe formatados em uma
47   * String.
48   * @return uma String contendo uma representação dos campos desta classe.
49   */
50   public String toString()
51     {
52     // Retorna diretamente uma String criada com os valores dos campos.
53     return "Título            : "+título+"\n"+
54            "Autor             : "+autor+"\n"+
55            "Número de Páginas : "+númeroDePáginas+"\n"+
56            "Ano da Edição     : "+anoDaEdição+"\n";
57     }
58
59   } // fim da classe Livro
```

A classe `Livro` contém somente os métodos mais essenciais para a demonstração de herança múltipla: o construtor, alguns métodos que permitem o acesso a alguns dos campos e o método `toString`.

Finalmente, na listagem 9.15, mostramos a classe `LivroDeBiblioteca` que demonstra herança múltipla herdando os campos e métodos públicos da classe `Livro` e implementando os métodos declarados na interface `ItemDeBiblioteca`.

Listagem 9.15: A classe `LivroDeBiblioteca`, que encapsula os dados de um livro de uma biblioteca.

```
1  /**
2   * A classe LivroDeBiblioteca, que encapsula os dados de um livro de uma biblioteca.
3   * Esta classe herda todos os campos e métodos da classe Livro (consideramos que campos
4   * privados são herdados, mesmo que não possam ser acessados diretamente), e implementa
5   * os métodos declarados na interface ItemDeBiblioteca.
6   */
7  class LivroDeBiblioteca extends Livro implements ItemDeBiblioteca
8    {
9   /**
10   * Declaração dos campos da classe
11   */
12   // Este campo que indica o status do empréstimo do material, e é usado por vários
13   // métodos, inclusive os que são implementados por causa da interface
14   // ItemDeBiblioteca
15   private boolean estáEmprestado;
16   // Este campo representa a localização do livro na biblioteca.
17   private String localização;
18
19  /**
20   * O construtor para a classe LivroDeBiblioteca, que recebe argumentos para
21   * inicializar todos os campos da classe. Este é o construtor "completo". Este
22   * construtor passará para o construtor da classe ancestral a tarefa de inicializar
23   * os campos que são herdados da classe ancestral.
24   * @param tit o título do livro
25   * @param aut o(s) nome(s) do(s) autor(es) do livro
26   * @param np o número de páginas do livro
27   * @param ae o ano da edição do livro
28   * @param loc a localização do livro
29   */
30   LivroDeBiblioteca(String tit,String aut,short np,short ae,String loc)
31     {
32     super(tit,aut,np,ae); // delega a inicialização ao construtor da classe ancestral
33     localização = loc;
34     estáEmprestado = false; // se acabou de ser criado, não deve estar emprestado ainda
35     }
36
37  /**
38   * O método empresta deve ser implementado porque foi declarado na interface
39   * ItemDeBiblioteca. Este método modifica o valor do campo estáEmprestado para true.
40   */
41   public void empresta()
42     {
43     estáEmprestado = true;
44     System.out.print("O livro "+qualTítulo()+" foi emprestado, e deverá ser ");
45     System.out.println("devolvido em "+máximoDeDiasParaEmpréstimo+" dias.");
46     }
47
48  /**
49   * O método devolve deve ser implementado porque foi declarado na interface
50   * ItemDeBiblioteca. Este método modifica o valor do campo estáEmprestado para false.
51   */
52   public void devolve()
53     {
54     estáEmprestado = false;
55     }
56
57  /**
58   * O método estáEmprestado deve ser implementado porque foi declarado na interface
59   * ItemDeBiblioteca. Este método retorna o valor do campo privado estáEmprestado.
60   * @return o valor do campo estáEmprestado
61   */
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
62    public boolean estáEmprestado()
63      {
64      return estáEmprestado;
65      }
66
67    /**
68     * O método localização deve ser implementado porque foi declarado na interface
69     * ItemDeBiblioteca. Este método retorna o valor do campo privado localização.
70     * @return o valor do campo localização
71     */
72    public String localização()
73      {
74      return localização;
75      }
76
77    /**
78     * O método descriçãoSumária deve ser implementado porque foi declarado na interface
79     * ItemDeBiblioteca.
80     * @return uma descrição sumária do livro.
81     */
82    public String descriçãoSumária()
83      {
84      return qualTítulo()+" "+qualAutor();
85      }
86
87    /**
88     * O método toString retorna os valores dos campos desta classe formatados em uma
89     * String. Este método chama explicitamente o método toString da classe ancestral.
90     * @return uma String contendo uma representação dos campos desta classe.
91     */
92    public String toString()
93      {
94      // Cria uma String temporária para receber o conteúdo dos campos. Esta String
95      // receberá primeiramente o resultado do método toString da classe ancestral.
96      String temporária = super.toString();
97      temporária += "Localização       : "+localização+"\n";
98      if (estáEmprestado)
99        temporária += "Este livro está emprestado.";
100     else
101       temporária += "Este livro está disponível para empréstimo.";
102     return temporária;
103     }
104
105   } // fim da classe LivroDeBiblioteca
```

Alguns pontos de interesse da classe `LivroDeBiblioteca` e da técnica de herança múltipla usando classes ancestrais e interfaces são:

- A classe herda alguns métodos da classe `Livro` que são usados indiretamente, como por exemplo, o construtor (que é chamado do construtor da classe `LivroDeBiblioteca` e o método `toString`; e métodos que são herdados e podem ser usados diretamente (como `qualTítulo` e `qualAutor`).
- Podemos ver nos exemplos um pequeno inconveniente da técnica de herança via implementação de interfaces: idealmente, qualquer item de biblioteca deveria ter alguns campos que representassem a localização do item na biblioteca, que mostrassem se o item está emprestado ou não, que representassem a data de aquisição do item, etc.
  Infelizmente, campos em interfaces são implicitamente `final`, significando que não podemos modificar estes campos. A consequência disto é que não podemos colocar nas interfaces os campos que uma classe que implementa esta interface pode manipular - os métodos são declarados na interface e os campos que tem relação com estes métodos na classe herdeira.

Finalmente, a classe `DemoLivroDeBiblioteca` (listagem 9.16) demonstra o uso de instâncias da classe `LivroDeBiblioteca`.

Listagem 9.16: A classe `DemoLivroDeBiblioteca`, que demonstra o uso de instâncias da classe `LivroDeBiblioteca`.

```
/**
 * A classe DemoLivroDeBiblioteca, que demonstra o uso de instâncias da classe
 * LivroDeBiblioteca.
 */
class DemoLivroDeBiblioteca // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * instâncias da classe LivroDeBiblioteca e uso de alguns de seus métodos.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Declaramos e criamos algumas instâncias da classe LivroDeBiblioteca
    LivroDeBiblioteca genesvii =
        new LivroDeBiblioteca("Genes VII","Benkamin Lewin",(short)990,(short)2000,
                              "Estante L, prateleira 3");
    LivroDeBiblioteca essentialCellBiology =
        new LivroDeBiblioteca("Essential Cell Biology","Bruce Alberts, Dennis Bray, "+
                                                       "Alexander Johnson, "+
                                                       "Julian Lewis, Martin Raff, "+
                                                       "Keith Roberts, Peter Walter",
                              (short)740,(short)1997,
                              "Estante A, prateleira 1");
    // Emprestamos um dos livros
    genesvii.empresta();
    // Imprimimos os dados dos livros
    System.out.println(genesvii);
    System.out.println(essentialCellBiology);
    } // fim do método main

  } // fim da classe DemoLivroDeBiblioteca
```

### 9.4.1 Conflitos de herança múltipla

*conflito de campos*

Existe um problema potencial com herança múltipla, que ocorrerá quando uma classe deve implementar mais de uma interface, e duas ou mais destas interfaces declaram campos com o mesmo nome - a classe que implementa os métodos não poderá ser compilada por causa de um conflito de nomes. Embora seja um problema pouco frequente, e que possa ser facilmente solucionado se as classes e interfaces envolvidas forem reescritas, ele merece uma menção neste capítulo.

Podemos exemplificar este problema considerando a hierarquia das classes e interfaces que representam itens em uma biblioteca, mostrada anteriormente neste capítulo. Consideremos que alguns itens de biblioteca são considerados como raros, e devem ter cuidados adicionais quando forem emprestados, além de ter um prazo de empréstimo diferente. Para definir métodos em classes que encapsulam itens raros de biblioteca, a interface `ItemRaroDeBiblioteca` (listagem 9.17) foi criada.

Listagem 9.17: A interface `ItemRaroDeBiblioteca`, que declara os campos e métodos que itens raros de uma biblioteca devem ter.

```
/**
 * A interface ItemRaroDeBiblioteca, que declara os campos e métodos que qualquer item
 * em uma biblioteca deve ter.
```

```
 */
interface ItemRaroDeBiblioteca
  {
 /**
  * Declaração dos campos da interface - note que todo campo em uma interface é
  * considerado como public final, não podendo variar.
  */
  int máximoDeDiasParaEmpréstimo = 1;
  boolean empréstimoPodeSerRenovado = false;

 /**
  * O método cuidadosEspeciais retorna uma String contendo uma descrição dos cuidados
  * especiais que devem ser tomados ao manusear este item
  * @return os cuidados especiais a tomar com este item
  */
  String cuidadosEspeciais();

  } // fim da interface ItemRaroDeBiblioteca
```

Outra classe que será útil para exemplificar o problema de conflitos em herança múltipla é a classe `Mapa`, que declara campos e métodos de um mapa (visto como um documento), e que será usada como classe ancestral da classe que contém o erro de conflito de campos. A classe `Mapa` é mostrada na listagem 9.18.

Listagem 9.18: A classe `Mapa`, que encapsula os dados de um mapa.

```
/**
 * A classe Mapa, que encapsula os dados de um mapa (considerando mapas como
 * tipos de documentos em bibliotecas, por exemplo). Para simplificar, somente uns
 * campos e métodos mais básicos são implementados nesta classe.
 */
class Mapa
  {
 /**
  * Declaração dos campos da classe
  */
  private String título; // o título do mapa
  private float escala; // a escala em que o mapa está
  private String descriçãoSumária; // uma descrição do mapa

 /**
  * O construtor para a classe Mapa, que recebe argumentos para inicializar todos
  * os campos da classe. Este é o construtor "completo".
  * @param tit o título do livro
  * @param aut o(s) nome(s) do(s) autor(es) do livro
  * @param np o número de páginas do livro
  * @param ae o ano da edição do livro
  */
  Mapa(String tit,float esc,String ds)
    {
    título = tit; escala = esc; descriçãoSumária = ds;
    }

 /**
  * O método qualTítulo retorna o título deste mapa.
  * @return o título do mapa
  */
 public String qualTítulo()
   {
   return título;
   }

 /**
  * O método descriçãoSumária retorna a descrição sumária deste mapa.
  * @return a descrição descrição sumária do mapa
  */
```

```
41  public String descriçãoSumária()
42    {
43    return descriçãoSumária;
44    }
45
46  /**
47   * O método toString retorna os valores dos campos desta classe formatados em uma
48   * String.
49   * @return uma String contendo uma representação dos campos desta classe.
50   */
51   public String toString()
52     {
53     // Retorna diretamente uma String criada com os valores dos campos.
54     return "Título           : "+título+"\n"+
55            "Escala           : "+escala+"\n"+
56            "Descrição Sumária: "+descriçãoSumária+"\n";
57     }
58
59   } // fim da classe Mapa
```

Finalmente, criamos a classe `MapaDeBiblioteca`, que herda da classe `Mapa` e implementa as interfaces `ItemDeBiblioteca` (por causa de métodos básicos de empréstimo e devolução de itens) e `ItemRaroDeBiblioteca`. Consideramos que mapas representados por esta classe são por *default*, raros (possivelmente bibliotecas tem somente um exemplar). A classe `MapaDeBiblioteca` é mostrada na listagem 9.19.

Listagem 9.19: A classe `MapaDeBiblioteca`, que encapsula os dados de um mapa de uma biblioteca.

```
1  /**
2   * A classe MapaDeBiblioteca, que encapsula os dados de um mapa de uma biblioteca.
3   * Esta classe herda todos os campos e métodos da classe Mapa (consideramos que campos
4   * privados são herdados, mesmo que não possam ser acessados diretamente), e implementa
5   * os métodos declarados na interface ItemDeBiblioteca.
6   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
7   */
8  class MapaDeBiblioteca extends Mapa implements ItemDeBiblioteca,ItemRaroDeBiblioteca
9    {
10   /**
11    * Declaração dos campos da classe
12    */
13   // Este campo que indica o status do empréstimo do material, e é usado por vários
14   // métodos, inclusive os que são implementados por causa da interface
15   // ItemDeBiblioteca
16   private boolean estáEmprestado;
17   // Este campo representa a localização do mapa na biblioteca - o campo é constante.
18   private final String localização = "Mapoteca";
19
20   /**
21    * O construtor para a classe MapaDeBiblioteca, que recebe argumentos para
22    * inicializar todos os campos da classe. Este é o construtor "completo". Este
23    * construtor passará para o construtor da classe ancestral a tarefa de inicializar
24    * os campos que são herdados da classe ancestral.
25    * @param tit o título do mapa
26    * @param esc a escala do mapa
27    * @param ds uma descrição sumária do mapa
28    */
29   MapaDeBiblioteca(String tit,float esc,String ds)
30     {
31     super(tit,esc,ds); // delega a inicialização ao construtor da classe ancestral
32     estáEmprestado = false; // se acabou de ser criado, não deve estar emprestado ainda
33     }
34
35   /**
36    * O método empresta deve ser implementado porque foi declarado na interface
37    * ItemDeBiblioteca. Este método modifica o valor do campo estáEmprestado para true.
```

```
38    */
39    public void empresta()
40      {
41      estáEmprestado = true;
42      System.out.print("O mapa "+qualTítulo()+" foi emprestado, e deverá ser ");
43      System.out.println("devolvido em "+máximoDeDiasParaEmpréstimo+" dias.");
44      }
45
46    /**
47     * O método devolve deve ser implementado porque foi declarado na interface
48     * ItemDeBiblioteca. Este método modifica o valor do campo estáEmprestado para false.
49     */
50    public void devolve()
51      {
52      estáEmprestado = false;
53      }
54
55    /**
56     * O método estáEmprestado deve ser implementado porque foi declarado na interface
57     * ItemDeBiblioteca. Este método retorna o valor do campo privado estáEmprestado.
58     * @return o valor do campo estáEmprestado
59     */
60    public boolean estáEmprestado()
61      {
62      return estáEmprestado;
63      }
64
65    /**
66     * O método localização deve ser implementado porque foi declarado na interface
67     * ItemDeBiblioteca. Este método retorna o valor do campo privado localização.
68     * @return o valor do campo localização
69     */
70    public String localização()
71      {
72      return localização;
73      }
74
75    /**
76     * O método descriçãoSumária deve ser implementado porque foi declarado na interface
77     * ItemDeBiblioteca.
78     * @return uma descrição sumária do mapa
79     */
80    public String descriçãoSumária()
81      {
82      return qualTítulo()+" ("+super.descriçãoSumária()+")";
83      }
84
85    /**
86     * O método cuidadosEspeciais deve ser implementado porque foi declarado na interface
87     * ItemRaroDeBiblioteca.
88     * @return os cuidados especiais a serem tomados com este item
89     */
90    public String cuidadosEspeciais()
91      {
92      return "Não amassar, dobrar ou tirar cópias heliográficas.";
93      }
94
95    /**
96     * O método toString retorna os valores dos campos desta classe formatados em uma
97     * String. Este método chama explicitamente o método toString da classe ancestral.
98     * @return uma String contendo uma representação dos campos desta classe.
99     */
100   public String toString()
101     {
102     // Cria uma String temporária para receber o conteúdo dos campos. Esta String
103     // receberá primeiramente o resultado do método toString da classe ancestral.
104     String temporária = super.toString();
105     temporária += "Localização      : "+localização+"\n";
106     if (estáEmprestado)
```

```
107       temporária += "Este mapa está emprestado.\n";
108     else
109       temporária += "Este mapa está disponível para empréstimo.\n";
110     temporária += "Cuidados especiais: "+cuidadosEspeciais();
111     return temporária;
112     }
113
114   } // fim da classe MapaDeBiblioteca
```

Os pontos interessantes da classe `MapaDeBiblioteca` (listagem 9.19) em especial em relação à sua classe ancestral `Mapa` (listagem 9.18) e à interface `ItemRaroDeBiblioteca` (listagem 9.17) são:

- Múltiplas interfaces podem ser implementadas. Na declaração de que interfaces uma classe implementa, podemos separar os vários nomes de interfaces por vírgulas. A ordem de declaração de implementação das interfaces é indiferente.
- Ambas as interfaces `ItemDeBiblioteca` e `ItemRaroDeBiblioteca` declaram um campo chamado `máximoDeDiasParaEmpréstimo`. Como ambas as interfaces são implementadas pela classe `MapaDeBiblioteca`, o campo é ambíguo: quando a classe for compilada, qual dos valores (que são `public` e `final`) deverá ser usado ? Como o compilador não pode escolher um dos dois em detrimento do outro, ele mostra uma mensagem de erro e não compila a classe.

*ambiguidade de campos*

- A classe `MapaDeBiblioteca` implementa um método `descriçãoSumária` que sobrepõe a declaração do mesmo método na classe `Mapa`, e implementa o método declarado na interface `ItemDeBiblioteca`. Isto não caracteriza conflito de nomes, uma vez que a regra de implementação de métodos de interfaces foi atendida.
- Se duas ou mais interfaces declararem um método com a mesma assinatura e uma classse implementar estas interfaces, não haverá conflito de nomes, e somente uma versão do método deverá ser implementada. Em outras palavras, múltiplas declarações de métodos em diversas interfaces não obrigam a classe que as implementa a definir várias cópias do método (mesmo porque isto iria contra as regras de sobrecarga de métodos).

## 9.5 Exercícios do capítulo 9

**Exercício 9.1:** ⋆

O que ocorreria se na declaração da classe `RoboAbstrato` (listagem 9.1) a palavra-chave `abstract` fosse omitida ? Explique.

**Exercício 9.2:** ⋆

Explique com suas palavras por que uma classe abstrata não pode ser instanciada.

**Exercício 9.3:** ⋆

Escreva um método `recarrega` para a classe `RoboABateria` (listagem 9.3) que recarregue o robô, isto é, que acumule ao seu campo `energia` um valor passado como argumento para o método.

**Exercício 9.4:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Produto
2    {
3    private String identificação;
4    private double custoDeFabricação;
5    Produto(String i,double c)
6      {
7      identificação = i; custoDeFabricação = c;
8      }
9    abstract public String toString();
10   abstract public void novoCusto(double nc);
11   }
```

**Exercício 9.5:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  abstract class Livro
2    {
3    abstract private String título;
4    abstract private String autor;
5    Livro(String t,double a);
6      {
7      título = t; autor = a;
8      }
9    abstract public String toString();
10   }
```

**Exercício 9.6:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  abstract class Pessoa
2    {
3    private String nome;
4    Pessoa(String n)
5      {
6      nome = n;
7      }
8    abstract public String toString()
9      {
10     return "O nome é "+nome;
11     }
12   }
```

**Exercício 9.7:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  abstract class Dispositivo
2    {
3    private String nome;
4    private long capacidadeEmBytes;
5    Dispositivo(String n,long c)
6      {
7      nome = n;  capacidadeEmBytes = c;
8      }
9    abstract public String toString();
10   abstract public double capacidadeEmMegabytes();
11   }
12
13 class DiscoOtico extends Dispositivo
14   {
15   DiscoOtico(long c)
16     {
17     super("Disco Ótico",241172480L);
18     }
19   public String toString()
20     {
21     return "Dispositivo:"+nome+" Capacidade:"+c;
22     }
23   }
```

**Exercício 9.8:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class Elemento
2    {
3    private String nome;
4    private double pesoAtômico;
5    Elemento(String n,double p)
6      {
7      nome = n; pesoAtômico = p;
8      }
9    public String toString()
10     {
11     return formato();
12     }
13   abstract private String formato();
14   }
```

**Exercício 9.9:** ⋆

Explique com suas palavras por que não podemos ter construtores declarados com a palavra-chave `abstract`.

**Exercício 9.10:** ⋆

Explique com suas palavras por que uma interface não pode ter métodos estáticos.

**Exercício 9.11:** ⋆

Explique, com suas palavras, porque interfaces não podem ter construtores.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 9.12:** ⋆

O que aconteceria se tentássemos chamar o método `imprimeTodosOsDados` da classe `DemoObjetosGeometricos` (listagem 9.8) passando para ele, como argumento, uma String ? Explique.

**Exercício 9.13:** ⋆

O que aconteceria se no método `main` da classe `DemoObjetosGeometricosEPolimorfismo` (listagem 9.9) escrevêssemos a linha `o1 = new ObjetoGeometrico();` depois da declaração da referência `o1` ? Explique.

**Exercício 9.14:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  interface Cor
2    {
3    String red   = "#FF0000";
4    String green = "#00FF00";
5    String blue  = "#0000FF";
6    String white = "#FFFFFF";
7    String black = "#000000";
8    }
9
10 interface NivelDeCinza
11   {
12   String white = "#FFFFFF";
13   String gray  = "#7F7F7F";
14   String black = "#000000";
15   }
16
17 class TextoHTML implements Cor,NivelDeCinza
18   {
19   private String texto;
20   final private char aspas = '"';
21   void imprime()
22     {
23     imprime(black);
24     }
25   void imprime(String cor)
26     {
27     System.out.print("<font color="+aspas+cor+aspas+">");
28     System.out.print(texto);
29     System.out.print("</font>");
30     }
31   }
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 9.15:** ⋆⋆

A classe `RoboSimples` (listagem 9.2) representa um robô que pode se movimentar apenas nas quatro direções cardeais, mas não impõe restrições no valor do ângulo que pode ser passado para o método `mudaDireção` (herdado da classe `RoboAbstrato`). Escreva, para a classe `RoboSimples`, o método `mudaDireção`, sobreponto o método herdado, de forma que se a direção atual seja calculada como:

- Se valores abaixo de 45 graus ou acima de 315 graus forem passados como argumento para o método, a direção do robô será 0 (zero) graus;
- Se valores entre 45 e 135 graus forem passados como argumento para o método, a direção do robô será 90 graus;
- Se valores entre 135 e 225 graus forem passados como argumento para o método, a direção do robô será 180 graus;
- Se valores entre 225 e 315 graus forem passados como argumento para o método, a direção do robô será 270 graus.

**Exercício 9.16:** ⋆⋆

Modifique o método `move` da classe `RoboABateria` (listagem 9.3) para que a energia gasta para movimentos na diagonal (45, 135, 225 e 315 graus) seja igual a 14 vezes o número de passos, sendo que para os movimentos na horizontal ou vertical deve continuar igual a 10 vezes o número de passos.

**Exercício 9.17:** ⋆⋆

Podemos declarar um método em uma interface como `private`, `static` ou `final` ? O que ocorrerá ? Explique cada um dos casos.

**Exercício 9.18:** ⋆⋆

Reescreva a classe `ConstantesMatematicas` (listagem 5.4) como uma interface. Demonstre seu uso através de uma classe que implementa esta interface e tem acesso aos seus campos.

**Exercício 9.19:** ⋆⋆

Escreva a classe `Quadrado`, que implementa a interface `ObjetoGeometrico`. Use como referência as classes `Circulo` e `Retangulo` (listagens 9.6 e 9.7).

**Exercício 9.20:** ⋆⋆

Escreva, para a classe `Retangulo` (listagem 9.7), o método `calculaAchatamento`, que calcula o achatamento do retângulo (o achatamento é dado dividindo-se o comprimento do lado menor pelo comprimento do lado maior).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 9.21:** ⋆⋆

Usando a interface `ObjetoGeometrico` (listagem 9.5) como base, crie a interface `ObjetoTridimensional` que deverá ser implementada por todas as classes que representem objetos tridimensionais. A interface deve declarar os métodos `centro` (que deve retornar uma instância da classe `Ponto3D`), `calculaSuperfície` que calcule e retorne a área de superfície do objeto tridimensional e `calculaVolume` que calcule e retorne o volume do objeto tridimensional.

**Exercício 9.22:** ⋆⋆

Inclua, na interface `ObjetoGeometrico` (listagem 9.5), a declaração do método `public String toString`, que fará com que todos as classes que implementarem a interface `ObjetoGeometrico` sejam obrigadas a implementar o método `toString`. Deliberadamente, deixe de implementar este método em uma classe que implemente a interface (pode-se usar como exemplo a classe `Circulo` ou `Retangulo`, retirando-se delas o método `toString`). Escreva uma aplicação que use uma destas classes. O que acontece se o método `toString` não for implementado apesar de estar declarado na interface ? Explique.

**Exercício 9.23:** ⋆⋆

Podemos ter na interface `Escalavel` um método `reduz` que chame o método `amplia` com valores entre zero e um ? Explique.

**Exercício 9.24:** ⋆⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  interface Resetavel
2    {
3    void reseta();
4    }
5
6  interface Modificavel
7    {
8    void reseta(int origem);
9    void modifica(int tamanho);
10   }
11
12 class Contador implements Resetavel,Modificavel
13   {
14   int valor;
15   void reseta()
16     {
17     tamanho = 1;
18     }
19   void modifica(int tam)
20     {
21     tamanho = tam;
22     }
23   }
```

**Exercício 9.25:** ⋆⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  interface Resetavel
2    {
3    void reseta();
4    }
5
6  interface Modificavel
7    {
8    void reseta();
9    void zera();
10   }
11
12 class Contador implements Resetavel,Modificavel
13   {
14   final int valor;
15   void reseta() // para interface Resetavel
16     {
17     valor = 1;
18     }
19   void reseta() // para interface Modificavel
20     {
21     valor = 1;
22     }
23   void zera()
24     {
25     valor = 0;
26     }
27   }
```

**Exercício 9.26:** ⋆⋆⋆

Escreva uma classe `RoboComMemoria` que herde da classe `RoboAbstrato` e que seja capaz de armazenar o número de passos dados em cada direção cardeal. Escreva, para este robô, um método `retornaÀOrigem` que, usando os passos dados em cada direção e a posição atual, calcule a sua origem. *Dica:* se o robô somente se move nas direções cardeais, não é necessário armazenar quatro valores para a memória do movimento, pois cada passo dado em uma direção corresponde ao valor negativo desses passos dados na direção oposta.

**Exercício 9.27:** ⋆⋆⋆

Escreva uma classe `RoboPesadoABateria` que herde da classe `RoboABateria` e que represente também o peso do robô em quilos. Este peso determinará a constante a ser usada para o gasto de energia com o movimento do robô: para que o robô se movimente na direção horizontal ou vertical o gasto de energia será o peso do robô vezes o número de passos a ser percorrido, e para que o robô se movimente na diagonal, o gasto de energia será 1.4 vezes o peso do robô vezes o número de passos a ser percorrido. Veja também o exercício 9.16.

**Exercício 9.28:** ⋆⋆⋆

Escreva para a interface `ObjetoGeometrico` (listagem 9.5) a declaração do método `clona`, que não recebe argumentos e retorna uma instância de `ObjetoGeometrico`. Ao criar esta declaração, todas as classes que implementam `ObjetoGeometrico` deverão implementar este método. Crie este método nas classes que implementam a interface. *Dica:* o método deverá ter o mesmo tipo de retorno do declarado na interface. Por exemplo, a classe `Circulo` deverá ter um método `public ObjetoGeometrico clona`, que deve criar e retornar um clone do círculo sendo encapsulado. Não existe problema em declarar um método como devendo retornar `ObjetoGeometrico` e retornando `Circulo`, já que por causa das regras de polimorfismo, `Circulo` *é-um-tipo-de* `ObjetoGeometrico`.

**Exercício 9.29:** ⋆⋆⋆

Reescreva a classe `Lampada` como sendo uma interface. Escreva as classes `LampadaIncandescente` e `LampadaDeNatal` que implementam a interface `Lampada`. Faça com que a classe `LampadaDeNatal` implemente também a classe `Cor`, que contém constantes representando diversas cores padrão.

**Exercício 9.30:** ⋆⋆⋆

Escreva a classe `Triangulo`, que implementa a interface `ObjetoGeometrico`. Use como referência as classes `Circulo` e `Retangulo` (listagens 9.6 e 9.7). *Dica:* para calcular o perímetro e a área de um triângulo qualquer, devemos obter os lados $a$, $b$ e $c$ do triângulo. Seu perímetro será calculado como $a+b+c$ e sua área como $\sqrt{s(s-a)(s-b)(s-c)}$, onde $s$ é a metade do perímetro. A raiz quadrada pode ser calculada com o método `Math.sqrt`. Para obter os lados, deve-se tomar os pontos do triângulo dois a dois e calcular o lado com a fórmula $l_{AB} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$ onde $l_{AB}$ é o lado que une os pontos $A$ e $B$.

**Exercício 9.31:** ⋆⋆⋆

Considere a classe `DemoObjetosGeometricos` (listagem 9.8). Modifique o seu método `imprimeTodosOsDados` para que este imprima também o achatamento dos retângulos (veja exercício 9.20).

**Exercício 9.32:** ⋆⋆⋆

Escreva a classe `Esfera`, que implemente a interface `ObjetoTridimensional` (exercício 9.21) e represente uma esfera no espaço tridimensional. O volume de uma esfera é igual a $\frac{4}{3}\pi r^3$, onde $r$ é o raio da esfera, e a área de sua superfície é igual a $4\pi r^2$.

**Exercício 9.33:** ⋆⋆⋆

Escreva a classe `Paralelepipedo`, que implemente a interface `ObjetoTridimensional` (exercício 9.21) e represente um paralelepípedo no espaço tridimensional. O volume de um paralelepípedo é igual a $a \times b \times c$, onde $a$, $b$ e $c$ são o comprimento de seus lados. A área da superfície é dada por $2(ab+bc+ac)$. A classe deve usar duas instâncias da classe `Ponto3D` para representar os pontos opostos do paralelepípedo. Os comprimentos dos lados podem ser obtidos a partir das diferenças das coordenadas $x$, $y$ e $z$ dos cantos opostos. Veja também a classe `Retangulo` (listagem 9.7) como exemplo.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 9.34:** ★★★

Escreva a classe `CilindroReto`, que implemente a interface `ObjetoTridimensional` (exercício 9.21) e represente um cilindro reto no espaço tridimensional. O volume de um cilindro reto é igual a $\pi r^2 h$, onde $r$ é o raio da base do cilindro e $h$ a sua altura. A área de sua superfície é igual a $2\pi rh + 2\pi r^2$.

**Exercício 9.35:** ★★★

Escreva a classe `ConeCircularReto`, que implemente a interface `ObjetoTridimensional` (exercício 9.21) e represente um cone circular reto no espaço tridimensional. O volume de um cone circular reto é igual a $\frac{1}{3}\pi r^2 h$, onde $r$ é o raio da base do cone e $h$ a sua altura. A área de sua superfície é igual a $\pi r\sqrt{r^2+h^2} + \pi r^2$.

**Exercício 9.36:** ★★★

Escreva a classe `TroncoDeConeCircularReto`, que implemente a interface `ObjetoTridimensional` (exercício 9.21) e represente um tronco de cone circular reto no espaço tridimensional. O volume de um tronco de cone circular reto é igual a $\frac{1}{3}\pi h(a^2 + ab + b^2)$, onde $a$ e $b$ são os raios da face inferior e superior e $h$ a sua altura. A área de sua superfície é igual a $\pi(a+b)\sqrt{h^2+(b-a)^2} + \pi a^2 + \pi b^2$.

**Exercício 9.37:** ★★★

Crie a classe `RetanguloEscalavel` que implementa simultaneamente as interfaces `ObjetoGeometrico` e `Escalavel`. Para simplificar o cálculo do novo tamanho do retângulo, considere que um dos pontos que o representa fica fixo e somente o outro tem seu tamanho modificado.

**Exercício 9.38:** ★★★★

Usando o resultado do exercício 9.26 e a classe `RoboABateria` (listagem 9.3) como base, escreva uma classe `RoboQueDeveVoltar` que implemente o mecanismo de retorno à posição de origem do robô e o mecanismo de consumo de energia dado. Escreva o método `move` de forma que o robô só se moverá se ainda houver energia suficiente para ele voltar ao ponto de origem. *Dica:* existem duas soluções para implementar o mecanismo de volta, uma delas é consideravelmente mais econômica (em termos de uso de energia do robô) do que a outra - tente implementar as duas soluções.

## 9.6 Exercícios complementares do capítulo 9

**Exercício 9.39:** ★

Modifique o construtor da classe `Circulo` (listagem 9.6) para que nenhum círculo com raio negativo possa ser criado. Caso um valor negativo seja passado como raio, o construtor deve considerar o raio como sendo zero.

**Exercício 9.40:** ★

Escreva um método `quantaEnergia` para a classe `RoboABateria` (listagem 9.3) que retorne a quantidade de energia restante para o robô.

**Exercício 9.41:** ⋆

Por que campos em interfaces devem ser inicializados em sua declaração ? Explique.

**Exercício 9.42:** ⋆

No método `toString` da classe `LivroDeBiblioteca` (listagem 9.15), usamos os campos `localização` e `estáEmprestado` ao invés dos métodos que permitem o acesso a estes campos. Existe alguma vantagem ou desvantagem em fazer isto ? Explique.

**Exercício 9.43:** ⋆

O que aconteceria, se no método `main` da classe `DemoLivroDeBiblioteca` (listagem 9.16), escrevêssemos a linha `essentialCellBiology.máximoDeDiasParaEmpréstimo = 45;` ? Explique.

**Exercício 9.44:** ⋆

Explique, com suas palavras, porque não faz sentido declararmos a classe `Livro` (listagem 9.14) como implementando a interface `ItemDeBiblioteca` (listagem 9.13).

**Exercício 9.45:** ⋆

O campo `empréstimoPodeSerRenovado`, na interface `ItemRaroDeBiblioteca` (listagem 9.17) não foi utilizado na classe `MapaDeBiblioteca` (listagem 9.19), que implementa a interface `ItemRaroDeBiblioteca`. Em que métodos da classe `MapaDeBiblioteca` este campo pode ser usado adequadamente ?

**Exercício 9.46:** ⋆⋆

Escreva um método `podeSeMover` para a classe `RoboABateria` (listagem 9.3) que retorne `true` caso o robô possa se mover na distância passada como argumento para o método, e `false` caso contrário. Escreva outro método sobrecarregado que não receba argumentos e considere que a distância a ser percorrida é um.

**Exercício 9.47:** ⋆⋆

Escreva a classe `FitaDeVideo`, que encapsula os dados de uma fita de vídeo educativo. Escreva também a classe `FitaDeVideoDeBiblioteca` que herda da classe `FitaDeVideo` e implementa a interface `ItemDeBiblioteca`. Use como referência a classe `LivroDeBiblioteca`, na listagem 9.15.

**Exercício 9.48:** ⋆⋆⋆

Modifique a classe abstrata `RoboAbstrato` (listagem 9.1) de forma que ao invés de usar dois valores separados para a posição do robô, a classe use uma instância da classe `Ponto2D`. Que modificações deverão ser feitas nas classes herdeiras ?

**Exercício 9.49:** ⋆⋆⋆

Modifique o método `amplia` da classe `RetanguloEscalavel` para que os dois pontos que definem o retângulo tenham suas posições modificadas. A figura 9.1 mostra duas maneiras de se modificar a escala de um retângulo, sendo que a primeira (ao centro) modifica somente um ponto, mantendo o outro fixo, enquanto que a segunda (à esquerda) modifica os dois pontos.

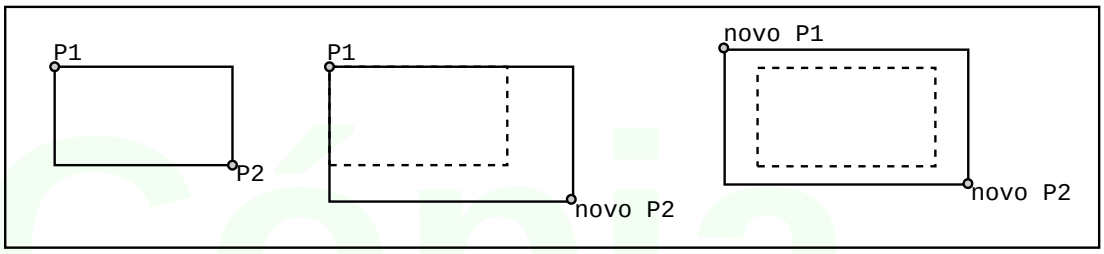



Figura 9.1: Duas maneiras de se modificar a escala de um retângulo

# Capítulo 10

# Pacotes de Classes em Java

## 10.1 Introdução

Vimos em exemplos anteriores que a criação de uma aplicação em Java, mesmo que seja simples, envolve a criação de várias classes. Mesmo que um único arquivo contendo os fontes das classes seja usado, para cada classe existente um arquivo com a extensão `.class` será criado. Claramente, para aplicações e projetos, é necessária uma organização das classes de forma que se saiba a qual aplicação ou projeto uma classe pertence.

Esta necessidade de organização fica ainda mais aparente quando se deseja compartilhar as classes ou instalá-las em um outro computador: sem um mecanismo de organização, seria necessário descobrir que classes são necessárias para a execução de uma aplicação qualquer, e a falta de uma classe poderia impedir a execução de toda a aplicação.

*pacotes de classes*

Java provê um mecanismo de agrupamento de classes em *pacotes* (em inglês, *packages*), com o qual podemos criar grupos de classes que mantém uma relação entre si. Para a criação destes pacotes, basta uma declaração de pertinência em cada classe e uma organização das classes em diretórios.

Até agora temos criado classes sem declarar a que pacote elas pertencem - todas as classes criadas sem declaração de pertinência em pacotes pertencem ao chamado pacote *default*. Como todas as classes criadas assim pertencem ao mesmo pacote, não são necessárias declarações adicionais quando usamos instâncias de uma classe dentro da outra - basta declarar as instâncias das outras classes que o compilador e máquina virtual se encarregarão de chamar os métodos destas classes.

Neste capítulo veremos como organizar classes em pacotes específicos, como usá-las e como os modificadores de acesso e mecanismo de herança se comportam quando pacotes são usados.

## 10.2 Criando pacotes de classes

*diretório de pacotes*

Pacotes requerem que as classes que comporão o pacote sejam armazenadas em um diretório específico[1]. A maneira mais simples de criar um pacote de classes é, então, criar um diretório e colocar lá todos os códigos-fonte das classes que serão consideradas como pertencentes àquele

[1] Este diretório pode ser qualquer diretório que esteja no caminho de procura de classes de Java. O apêndice B.1 mostra como podemos usar vários diretórios para armazenamento de classes e pacotes. Para finalidades práticas, consideraremos que os diretórios a que nos referimos sejam subdiretórios do diretório onde desenvolvemos os programas em Java

pacote.

Para exemplificar, vamos considerar as classes `Data`, `Hora` e a classe `DataHora`, que encapsula uma data e uma hora através do mecanismo de delegação. Para transformar estas classes, que claramente tem um propósito comum, em um pacote, primeiro devemos criar um diretório `DataHora` e armazenar as classes dentro deste diretório.

Cada classe pertencente a um pacote deve ter, no seu início, antes de qualquer outra declaração na classe, a palavra-chave `package` seguida do nome do diretório (e pacote) a qual esta classe deverá pertencer. A classe `Data`, mostrada na listagem 10.1, demonstra a declaração de pacote.

Listagem 10.1: A classe `Data`, que encapsula os dados de uma data qualquer e que faz parte do pacote `DataHora`.

```
1  /* Esta classe faz parte do pacote DataHora */
2  package DataHora;
3
4  /**
5   * A classe Data, que encapsula os dados de uma hora qualquer e que faz parte do
6   * pacote DataHora.
7   */
8  public class Data
9    {
10  /**
11   * Declaração dos campos da classe. Os campos são declarados sem modificadores,
12   * e serão acessíveis somente para as classes que herdarem desta.
13   */
14   byte dia,mês;
15   short ano;
16
17  /**
18   * O construtor da classe, que recebe argumentos para inicializar os campos da
19   * classe. Note que o construtor é declarado como sendo público, caso contrário
20   * ele não poderá ser chamado de fora do pacote !
21   * Nenhuma verificação é feita nos valores, para manter o exemplo simples.
22   * @param d o dia a ser encapsulado por uma instância desta classe
23   * @param m o mês a ser encapsulado por uma instância desta classe
24   * @param a o ano a ser encapsulado por uma instância desta classe
25   */
26   public Data(byte d,byte m,short a)
27     {
28     dia = d; mês = m; ano = a;
29     }
30
31  /**
32   * O método toString, que retorna uma String contendo os campos desta classe,
33   * formatados de forma adequada.
34   * @return uma String com os valores dos campos desta classe
35   */
36   public String toString()
37     {
38     return dia+"/"+mês+"/"+ano;
39     }
40
41   } // fim da classe Data
```

Alguns pontos interessantes na classe `Data` são:

- Antes da declaração da classe, uma declaração `package DataHora;` é feita, indicando que a classe fará parte do pacote `DataHora`. Para que as classes que usem esta classe sejam compiladas corretamente, devemos colocar a classe `Data` no diretório `DataHora` (exatamente o mesmo nome do pacote). `package`

Uma classe não pode pertencer a mais de um pacote - somente uma declaração de pacote é permitida por classe.

*classe pública*

- A classe é declarada explicitamente com o modificador `public`, o que garantirá que esta classe poderá ser usada (instanciada) a partir de qualquer outra classe, pertencente ou não ao pacote `DataHora`.
- O construtor da classe deve também ser declarado explicitamente com o modificador `public`, caso contrário não poderemos criar instâncias desta classe através do construtor.
- Os campos da classe foram declarados propositalmente sem modificadores. Note que isto não significa que os campos são públicos. Explicações mais detalhadas serão dadas na seção 10.3 deste capítulo.

A classe `Hora`, mostrada na listagem 10.2, também fará parte do pacote `DataHora`.

Listagem 10.2: A classe `Hora`, que encapsula os dados de uma hora qualquer e que faz parte do pacote `DataHora`.

```
1  /* Esta classe faz parte do pacote DataHora */
2  package DataHora;
3
4  /**
5   * A classe Hora, que encapsula os dados de uma hora qualquer e que faz parte do
6   * pacote DataHora.
7   */
8  public class Hora
9    {
10  /**
11   * Declaração dos campos da classe. Os campos são declarados sem modificadores,
12   * e serão acessíveis somente para as classes que herdarem desta.
13   */
14   byte hora,minuto,segundo;
15
16  /**
17   * O construtor da classe, que recebe argumentos para inicializar os campos da
18   * classe. Note que o construtor é declarado como sendo público, caso contrário
19   * ele não poderá ser chamado de fora do pacote !
20   * Nenhuma verificação é feita nos valores, para manter o exemplo simples.
21   * @param h a hora a ser encapsulada por uma instância desta classe
22   * @param m o minuto a ser encapsulado por uma instância desta classe
23   * @param s o segundo a ser encapsulado por uma instância desta classe
24   */
25   public Hora(byte h,byte m,byte s)
26     {
27     hora = h; minuto = m; segundo = s;
28     }
29
30  /**
31   * O método toString, que retorna uma String contendo os campos desta classe,
32   * formatados de forma adequada.
33   * @return uma String com os valores dos campos desta classe
34   */
35   public String toString()
36     {
37     return hora+":"+minuto+":"+segundo;
38     }
39
40   } // fim da classe Hora
```

Da mesma forma que na classe `Data`, a classe `Hora` e seu construtor são declarados explicitamente como `public`, e a classe é declarada como pertencente ao pacote `DataHora`. Os campos da classe também foram propositadamente declarados sem modificadores de acesso.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

A terceira classe pertencente ao pacote `DataHora` é a classe de mesmo nome, mostrada na listagem 10.3.

Listagem 10.3: A classe `DataHora`, que encapsula os dados de uma data e uma hora simultaneamente e que faz parte do pacote `DataHora`.

```
1  /* Esta classe faz parte do pacote DataHora */
2  package DataHora;
3
4  /**
5   * A classe DataHora, que reusa as classes Data e Hora através de delegação.
6   * A data e hora são representadas por instâncias das respectivas classes que estão
7   * embutidas na classe DataHora. Como as classes Data e Hora fazem parte do mesmo
8   * pacote e seus membros foram declarados sem modificadores de acesso, poderão ser
9   * acessados diretamente nesta classe.
10  */
11 public class DataHora // declaração da classe
12   {
13   /**
14    * Declaração dos campos da classe.
15    */
16   Data estaData; // uma instância da classe Data representa o dia, mês e ano
17   Hora estaHora; // uma instância da classe Hora representa a hora, minuto
18                  // e segundo
19
20  /**
21   * O construtor para a classe DataHora, que recebe argumentos para inicializar
22   * todos os campos que esta classe indiretamente contém, e chama os construtores
23   * das classes Data e Hora para inicializar os campos das instâncias destas classes.
24   * Note que o construtor é declarado como sendo público, caso contrário ele não
25   * poderá ser chamado de fora do pacote !
26   * @param h a hora
27   * @param min o minuto
28   * @param s o segundo
29   * @param d o dia
30   * @param m o mês
31   * @param a o ano
32   */
33   public DataHora(byte h,byte min,byte s,byte d,byte m,short a)
34     {
35     estaData = new Data(d,m,a);
36     estaHora = new Hora(h,min,s);
37     }
38
39   /**
40    * O método toString não recebe argumentos, e retorna uma String contendo os valores
41    * dos campos da classe formatados. Os campos das instâncias das classes Data e Hora
42    * podem ser acessados diretamente, já que foram declarados sem modificadores nas
43    * classes, e esta classe pertence ao mesmo pacote. Para demonstrar esta
44    * possibilidade, este método acessa diretamente os campos das classes Data e Hora.
45    * @return uma String com os valores dos campos formatados.
46    */
47   public String toString()
48     {
49     String resultado = estaHora.hora+":"+estaHora.minuto+":"+estaHora.segundo;
50     resultado += " de "+estaData.dia;
51     resultado += " de ";
52     switch(estaData.mês) // dependendo do valor do campo mês, concatena o nome do mês
53       {
54       case  1: resultado += "Janeiro"; break;
55       case  2: resultado += "Fevereiro"; break;
56       case  3: resultado += "Março"; break;
57       case  4: resultado += "Abril"; break;
58       case  5: resultado += "Maio"; break;
59       case  6: resultado += "Junho"; break;
60       case  7: resultado += "Julho"; break;
61       case  8: resultado += "Agosto"; break;
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
62        case  9: resultado += "Setembro"; break;
63        case 10: resultado += "Outubro"; break;
64        case 11: resultado += "Novembro"; break;
65        case 12: resultado += "Dezembro"; break;
66        }
67      resultado += " de "+estaData.ano;
68      return resultado;
69      }
70
71    } // fim da classe DataHora
```

Alguns pontos interessantes da classe `DataHora` são:

- Não existe nenhum problema em termos uma classe declarada com o mesmo nome do pacote - o nome do pacote indica a organização de um grupo de classes, e não conflita com nomes de classes.
- A classe `DataHora` encapsula uma instância da classe `Data` e da classe `Hora`. Como as classes `Data` e `Hora` pertencem à mesma classe que `DataHora`, a classe `DataHora` pode usar estas classes sem problemas ou necessidade de declarações adicionais. As instâncias das classes `Data` e `Hora` encapsuladas na classe `DataHora` também são declaradas sem modificadores.
- A classe `DataHora` e seu construtor também são declarados explicitamente com o modificador `public`.
- Mesmo podendo acessar diretamente os campos das classes `Data` e `Hora`, a inicialização dos campos é feita pelos construtores das respectivas classes. A razão é simples: as instâncias `estaData` e `estaHora` deverão ser obrigatoriamente alocadas com o operador `new`, que por sua vez executará o construtor das respectivas classes. Uma vez que não existe construtor *default* (vazio) para as classes `Data` e `Hora`, faz sentido que aproveitemos os construtores já existentes, que inicializam os campos da classe. Isto também faz sentido à luz dos exemplos de herança através de delegação ou composição, que foram vistos no capítulo 8.
- O método `toString` desta classe usa diretamente os campos das classes `Data` e `Hora` para formatar uma String que conterá os valores dos campos da classe. Os campos podem ser acessados pois foram declarados sem modificadores nas classes `Data` e `Hora`, sendo considerados públicos para o pacote `DataHora`.

As classes `Data`, `Hora` e `DataHora` pertencem, então, ao mesmo pacote, que deve estar localizado em um diretório `DataHora`. Para esclarecer melhor os efeitos da organização de classes em pacotes, a classe `DemoDataHora` (listagem 10.4) foi criada. Esta classe demonstra o uso das classes `Data`, `Hora` e `DataHora`, mas propositalmente não é parte do pacote `DataHora`.

Listagem 10.4: A classe `DemoDataHora`, que demonstra usos de instâncias das classes que fazem parte do pacote `DataHora`.

```
1  /* Esta classe precisa das classes que estão no pacote DataHora */
2  import DataHora.*;
3
4  /**
5   * A classe DemoDataHora, que demonstra usos de instâncias das classes que fazem
6   * parte do pacote DataHora. Esta classe declara, inicializa e usa algumas instâncias
7   * das classes Data, Hora e DataHora, mas tenta acessar campos destas classes que
8   * só podem ser acessados por classes do mesmo pacote.
9   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
10  */
11 class DemoDataHora
12   {
```

```
13  /**
14   * O método main permite a execução desta classe. Este método contém declarações de
15   * algumas instâncias das classes Data, Hora e DataHora.
16   * @param argumentos os argumentos que podem ser passados para o método via linha
17   *        de comando, mas que neste caso serão ignorados.
18   */
19   public static void main(String[] argumentos)
20     {
21     // Criamos uma instância da classe Hora
22     Hora meiodia = new Hora((byte)12,(byte)00,(byte)00);
23     // Criamos uma instância da classe Data
24     Data hoje  = new Data((byte)11,(byte)5,(short)2001);
25     // Criamos uma instância da classe DataHora
26     DataHora agora = new DataHora((byte)22,(byte)35,(byte)00,
27                                   (byte)11,(byte)5,(short)2001);
28     // Imprimimos as instâncias através de chamadas implícitas aos métodos toString
29     System.out.println(meiodia);
30     System.out.println(hoje);
31     System.out.println(agora);
32     // Tentamos mudar os campos das classes, que não foram declarados como private,
33     // mesmo assim causando erros pois a classe DemoDataHora não pertence ao mesmo
34     // pacote que as classes Data, Hora e DataHora.
35     meiodia.segundos = 17;
36     hoje.mês = 2;
37     DataHora.estaData.ano = 2000;
38     } // fim do método main
39
40   } // fim da classe DemoDataHora
```

Os pontos de interesse da classe `DemoDataHora` são:

- A classe não é declarada como pertencente a algum pacote, sendo considerada parte do pacote *default*.
- Todas as classes do pacote `DataHora` são importados (ou disponibilizados) para uso na classe `DemoDataHora`, através da declaração `import DataHora.*;` - o asterisco (`*`) indica que todas as classes daquele pacote devem ser importadas.
- O método `main` da classe `DemoDataHora` declara, inicializa e usa uma instância de cada uma das classes `Data`, `Hora` e `DataHora`, que mesmo não estando localizadas no mesmo diretório da classe `DemoDataHora` foram disponibilizadas pelo mecanismo de importação de pacotes.
- Alguns erros propositais foram incluidos na classe `DemoDataHora`: campos das classes `Data`, `Hora` e `DataHora`, que foram declarados sem modificadores, não podem ser acessados fora do pacote `DataHora`, mesmo que este tenha sido importado para uso na classe `DemoDataHora`.

Com os pontos mostrados por este pequeno exemplo de criação e uso de pacotes, podemos discutir mais detalhes sobre modificadores de acesso que ainda não foram discutidos e o real sentido do modificador *default*.

## 10.3 Pacotes e modificadores de acesso

Nesta seção veremos os outros modificadores de acesso de métodos e campos de classes em Java, e os efeitos de seu uso quando consideramos classes contidas em pacotes. Os modificadores `private` e `public`, cujo efeito é independente da existência e uso de pacotes, foram vistos na seção 2.6.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

### 10.3.1 O modificador *default*

*modificador* default

No exemplo mostrado pelas classes `Data`, `Hora` e `DataHora` (pertencentes ao pacote `DataHora`) e `DemoDataHora` vimos, em linhas gerais, o efeito do modificador *default*[2]: os campos e métodos declarados com o modificador *default* nas classes do pacote `DataHora` podem ser acessados sem problemas por outros métodos em outras classes no mesmo pacote. No exemplo dado, o método `toString` da classe `DataHora` usa diretamente campos que foram declarados sem modificadores nas classes `Data` e `Hora`.

Outras classes em outros pacotes, entretanto, não tem acesso aos campos declarados como *default*, como demonstrado pela classe `DemoDataHora`, que não pode ser compilada pois tenta acessar diretamente campos das classes `Data`, `Hora` e `DataHora`.

### 10.3.2 O modificador `protected`

*modificador* `protected`

O quarto e último modificador de acesso de Java é o modificador `protected` (protegido), que só fará sentido se usado em um contexto de pacotes. Campos e métodos declarados com este modificador podem ser usados diretamente por todas as classes pertencentes ao mesmo pacote.

*pacote* default

Se uma classe não é explicitamente declarada como pertencente a um pacote, o compilador considerará que ela pertence ao pacote *default*, e se nesta classe métodos e campos forem declarados como `protected`, eles poderão ainda assim ser acessados de todas as outras classes que pertençam ao pacote *default*, como se tivessem sido declarados como `public` ou sem modificador de acesso. Por esta razão não existe sentido em declararmos campos e métodos protegidos em classes que não pertençam a pacotes.

Adicionalmente, o modificador `protected` fará com que qualquer campo ou método declarado com ele possa ser acessado por qualquer classe que herde da classe onde os campos e/ou métodos foram declarados. Desta forma, `protected` é menos restrito do que *default* pois permite o acesso direto a campos herdados de classes de pacotes diferentes.

Para entender melhor os efeitos do uso do modificador `protected` em um pacote de classes em Java, consideremos um pequeno exemplo. O pacote `Pessoal` conterá as classes `Pessoa`, `Funcionario` e `ChefeDeDepartamento`, mostradas respectivamente nas listagens 10.5, 10.6 e 10.7.

Listagem 10.5: A classe `Pessoa`, que encapsula os dados de identificação de uma pessoa e que faz parte do pacote `Pessoal`.

```
1  /* Esta classe faz parte do pacote Pessoal */
2  package Pessoal;
3
4  /* Esta classe usa a classe Data, definida no pacote DataHora */
5  import DataHora.Data;
6
7  /**
8   * A classe Pessoa, que encapsula os dados de identificação de uma pessoa. Os campos
9   * desta classe são declarados como protected, podendo ser utilizados diretamente em
10  * classes herdeiras. A classe é declarada pública, para poder ser usada em outras
11  * classes fora do pacote Pessoal.
12  */
13 public class Pessoa // declaração da classe
14   {
```

[2] Para simplificar, consideremos o modificador *default* como um modificador de acesso como os outros, embora não exista uma palavra-chave que deva ser usada para a declaração de acesso *default*.

```
15  /**
16   * Declaração dos campos da classe
17   */
18   protected String nome;     // o nome da pessoa
19   protected int identidade; // o número da identidade da pessoa
20   protected Data nascimento; // a data de nascimento da pessoa
21
22  /**
23   * O construtor da classe Pessoa, que recebe argumentos para inicializar seus campos.
24   * @param n o nome da pessoa
25   * @param i o número da identidade da pessoa
26   * @param d a data de nascimento da pessoa
27   */
28   public Pessoa(String n,int i,Data d)
29     {
30     nome = n; identidade = i; nascimento = d;
31     }
32
33  /**
34   * O método toString não recebe argumentos, e retorna uma String contendo os valores
35   * dos campos da classe formatados.
36   * @return uma String com os valores dos campos formatados.
37   */
38   public String toString()
39     {
40     return "Nome: "+nome+"\nIdentidade: "+identidade+" "+
41            "\nData de Nascimento: "+nascimento;
42     }
43
44   } // fim da classe Pessoa
```

Alguns pontos interessantes na classe `Pessoa` (listagem 10.5) são:

- A classe é declarada como pertencente ao pacote `Pessoal`, devendo estar em um diretório com o mesmo nome.
- A classe importa a classe `Data` do pacote `DataHora` - note que a sintaxe indica que somente a classe `Data` será importada.
- A classe e seu construtor são declarados como sendo públicas (para permitir o acesso de fora do pacote).
- Os campos da classe são todos declarados como `protected`, podendo ser acessados diretamente por métodos em outras classes do mesmo pacote.

A classe `Funcionario` herda da classe `Pessoa` e declara alguns campos adicionais, e é mostrada na listagem 10.6.

Listagem 10.6: A classe `Funcionario`, que encapsula os dados básicos de um funcionário de uma empresa e que faz parte do pacote `Pessoal`.

```
1  /* Esta classe faz parte do pacote Pessoal */
2  package Pessoal;
3
4  /* Esta classe usa a classe Data, definida no pacote DataHora */
5  import DataHora.Data;
6
7  /**
8   * A classe Funcionario, que encapsula os dados básicos de um funcionário de uma
9   * empresa.  Os campos desta classe são declarados como protected, podendo ser
10  * utilizados diretamente em classes herdeiras. A classe é declarada pública, para
11  * poder ser usada em outras classes fora do pacote Pessoal.
12  * Esta classe herda da classe Pessoa, criando a relação "Funcionario é um tipo de
13  * Pessoa".
14  */
15 public class Funcionario extends Pessoa // declaração da classe (herdando de Pessoa)
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
16   {
17  /**
18   * Declaração dos campos adicionais da classe. Os campos da classe Pessoa serão
19   * herdados e poderão ser acessados diretamente pois foram declarados com o
20   * modificador protected na classe Pessoa.
21   */
22   protected Data admissão;
23   protected float salário;
24
25  /**
26   * O construtor para a classe Funcionario, que recebe argumentos para inicializar
27   * todos os campos da classe. Este construtor recebe os mesmos dados que seriam
28   * necessários para criar uma instância da classe Pessoa mais os que diferenciam
29   * a classe Funcionario da Pessoa. Passaremos os dados que representam uma Pessoa
30   * para o construtor da classe ancestral, via a palavra-chave super.
31   * @param nome o nome da pessoa
32   * @param id o número da identidade da pessoa
33   * @param nasc uma instância da classe Data com a data de nascimento
34   * @param adm uma instância da classe Data com a data de admissão
35   * @param sal o salário do funcionário
36   */
37   public Funcionario(String nome,int id,Data nasc,Data adm,float sal)
38     {
39     super(nome,id,nasc); // chama o construtor da classe Pessoa
40     admissão = adm;
41     salário = sal;
42     }
43
44  /**
45   * O método toString não recebe argumentos, e retorna uma String contendo os valores
46   * dos campos da classe formatados. Para demonstrar a acessibilidade aos campos da
47   * classe ancestral, este método usa diretamente o campo nome da classe Pessoa.
48   * @return uma String com os valores dos campos formatados.
49   */
50   public String toString()
51     {
52     String resultado;
53     resultado = "Nome: "+nome+"\n"; // uso direto de campo herdado
54     resultado = resultado + "Data de admissão: "+admissão+"\n";
55     resultado = resultado + "Salário: "+salário;
56     return resultado;
57     }
58
59   } // fim da classe Funcionario
```

Os pontos de interesse na classe `Funcionario` são:

- A classe `Funcionario` é declarada como pertencente ao pacote `Pessoal`, como pública e como herdeira da classe `Pessoa`.
- Já que a classe `Funcionario` usa instâncias da classe `Data`, ela deve importar a classe `Data` do pacote `DataHora` novamente - classes importadas para uso na classe `Pessoa` não são herdadas para a classe `Funcionario`.
- Os campos da classe também são declarados como `protected`. A classe `Funcionario` já tem acesso direto aos campos da classe `Pessoa`, que também foram declarados como `protected` - podemos ver isto no método `toString` da classe `Funcionario` que usa diretamente o campo `nome`, declarado como protegido na classe ancestral `Pessoa`.
- Curiosamente, precisamos chamar o construtor da classe ancestral de dentro do construtor da classe `Funcionario` usando a palavra-chave `super`. Seria de se esperar que já que os campos da classe `Pessoa` podem ser acessados diretamente de dentro da classe `Funcionario`, o construtor desta poderia inicializar diretamente todos os campos sem recorrer ao construtor da classe ancestral.
  Na verdade, o construtor de uma classe que herda explicitamente de outra sempre chama,



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

direta ou indiretamente o construtor da classe ancestral. Se a classe ancestral somente provê construtores com argumentos, estes devem ser chamados explicitamente a partir dos construtores das classes descendentes. Isto foi explicado com mais detalhes na seção 8.3.1.

A terceira classe do pacote `Pessoal` é a classe `ChefeDeDepartamento`, que herda da classe `Funcionario` e é mostrada na listagem 10.7.

Listagem 10.7: A classe `ChefeDeDepartamento`, que encapsula os dados básicos de um chefe de departamento de uma empresa e que faz parte do pacote `Pessoal`.

```
1  /* Esta classe faz parte do pacote Pessoal */
2  package Pessoal;
3
4  /* Esta classe usa a classe Data, definida no pacote DataHora */
5  import DataHora.Data;
6
7  /**
8   * A classe ChefeDeDepartamento, que encapsula os dados básicos de um chefe de um
9   * departamento de uma empresa. A classe é declarada pública, para poder ser usada em
10  * outras classes fora do pacote Pessoal. Esta classe herda da classe
11  * Funcionario, criando a relação "ChefeDeDepartamento é um tipo de Funcionario".
12  */
13 public class ChefeDeDepartamento extends Funcionario // declaração da classe (herdando
14                                                      // de Funcionario)
15   {
16  /**
17   * Declaração dos campos adicionais da classe. Somente os campos que diferenciam
18   * um ChefeDeDepartamento de um Funcionario são necessários. Os campos das classes
19   * Pessoa e Funcionario podem ser acessados diretamente dentro da classe.
20   */
21   protected String departamento;
22   protected Data promoçãoAChefe;
23
24  /**
25   * O construtor para a classe ChefeDeDepartamento, que recebe argumentos para
26   * inicializar todos os campos da classe. Este construtor recebe os mesmos dados
27   * que seriam necessários para criar uma instância da classe Pessoa mais os que
28   * diferenciam a classe Funcionario da classe Pessoa e os que diferenciam a classe
29   * ChefeDeDepartamento da classe Funcionario. Passaremos os dados que representam
30   * um Funcionario para o construtor da classe ancestral, via a palavra-chave super.
31   * @param nome o nome da pessoa
32   * @param id o número da identidade da pessoa
33   * @param nasc uma instância da classe Data com a data de nascimento
34   * @param adm uma instância da classe Data com a data de admissão
35   * @param sal o salário do funcionário/chefe de departamento
36   * @param dep o departamento que este chefe chefia
37   * @param prom uma instância da classe Data com a data de promoção ao cargo
38   */
39   public ChefeDeDepartamento(String nome,int id,Data nasc,
40                              Data adm,float sal,
41                              String dep,Data prom)
42     {
43     super(nome,id,nasc,adm,sal); // chama o construtor da classe Funcionario
44     departamento = dep;
45     promoçãoAChefe = prom;
46     }
47
48  /**
49   * O método toString não recebe argumentos, e retorna uma String contendo os valores
50   * dos campos da classe formatados.  Para demonstrar a acessibilidade aos campos das
51   * classes ancestrais, este método usa diretamente o campo nome da classe Pessoa e o
52   * campo salário da classe Funcionario.
53   * @return uma String com os valores dos campos formatados.
54   */
55   public String toString()
56     {
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
57      String resultado;
58      resultado = "Nome: "+nome+"\n"; // uso direto de campo herdado da classe Pessoa
59      resultado = resultado + "Departamento: "+departamento+"\n";
60      resultado = resultado + "Data de promoção ao cargo: "+promoçãoAChefe+"\n";
61      resultado = resultado + "Salário: "+salário; // uso direto de campo herdado da
62                                                   // classe Funcionario
63      return resultado;
64      }
65
66    } // fim da classe ChefeDeDepartamento
```

Similarmente à classe `Funcionario`, a classe `ChefeDeDepartamento` herda os campos declarados como `protected` das classes ancestrais, podendo usá-los diretamente como mostrado em seu método `toString`.

Finalmente, para demonstrar características e efeitos do modificador de acesso `protected`, usaremos a classe `DemoPessoal`, mostrada na listagem 10.8.

Listagem 10.8: A classe `DemoPessoal`, que demonstra usos de instâncias das classes que fazem parte do pacote `Pessoal`.

```
1  /* Esta classe precisa das classes que estão nos pacotes DataHora e Pessoal */
2  import DataHora.*;
3  import Pessoal.*;
4
5  /**
6   * A classe DemoPessoal, que demonstra usos de instâncias das classes que fazem
7   * parte do pacote Pessoal. Esta classe declara, inicializa e usa algumas instâncias
8   * das classes Pessoa, Funcionario e ChefeDeDepartamento. A classe mostra também que
9   * campos declarados como protected não podem ser acessados diretamente de classes
10  * não herdeiras.
11  * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
12  */
13 class DemoPessoal
14   {
15  /**
16   * O método main permite a execução desta classe. Este método contém declarações de
17   * algumas instâncias das classes Pessoa, Funcionario e ChefeDeDepartamento.
18   * @param argumentos os argumentos que podem ser passados para o método via linha
19   *       de comando, mas que neste caso serão ignorados.
20   */
21   public static void main(String[] argumentos)
22     {
23     // Criamos uma instância da classe Pessoa
24     Pessoa dawkins = new Pessoa("Richard Dawkins",
25                                 8275123,
26                                 new Data((byte)20,(byte)10,(short)1943));
27     // Criamos uma instância da classe Funcionario
28     Funcionario payne = new Funcionario("Roger Payne",
29                                         617653,
30                                         new Data((byte)12,(byte)7,(short)1933),
31                                         new Data((byte)17,(byte)3,(short)1956),
32                                         8241.0f);
33     // Criamos uma instância da classe ChefeDeDepartamento
34     ChefeDeDepartamento lorenz = new ChefeDeDepartamento("Konrad Lorenz",
35                                          981002,
36                                          new Data((byte)31,(byte)1,(short)1941),
37                                          new Data((byte)3,(byte)1,(short)1967),
38                                          18200.0f,
39                                          "Ciências do Comportamento",
40                                          new Data((byte)7,(byte)3,(short)1981));
41     // Imprimimos as três instâncias.
42     System.out.println(dawkins);
43     System.out.println(payne);
44     System.out.println(lorenz);
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
45      // Tentamos acessar campos das classes, o que não é permitido pelo compilador.
46      System.out.println(dawkins.nome);
47      System.out.println(payne.salário);
48      System.out.println(lorenz.departamento);
49      }
50
51    } // fim da classe DemoPessoal
```

Os pontos interessantes da classe `DemoPessoal` são:

- A classe utiliza classes dos pacotes `Pessoal` e `DataHora`. Para simplificar, todas as classes destes dois pacotes são importadas, com duas declarações `import` separadas. Não é possível importar todos os pacotes disponíveis com uma declaração do tipo `import *;`, nem agrupar pacotes a serem importados com uma única declaração `import`.
- O método `main` da classe declara, inicializa e imprime três instâncias das classes `Pessoa`, `Funcionario` e `ChefeDeDepartamento`, mas não pode ser compilada pois tenta acessar os campos destas classes que foram declarados como `protected` - estes campos só podem ser acessados por métodos de classes do mesmo pacote, e a classe `DemoPessoal` é implicitamente pertencente ao pacote *default*.

### 10.3.3 Sumário dos modificadores de acesso e de suas regras

*modificadores de acesso*

A tabela 10.1 sumariza as regras de acesso dependendo dos modificadores. Consideramos que um campo ou método pode ser declarado com um dos quatro modificadores mostrados nas colunas da tabela. Nas linhas da tabela estão mostradas as possibilidades de acesso dos campos ou métodos a partir de métodos da mesma e de outras classes.

| | `private` | `protected` | *default* | `public` |
|---|---|---|---|---|
| Métodos da mesma classe | sim | sim | sim | sim |
| Métodos de classes herdeiras | não | sim | sim | sim |
| Métodos de outras classes no mesmo pacote | não | não | sim | sim |
| Métodos de outras classes em outro pacote | não | não | não | sim |

Tabela 10.1: Efeito dos modificadores de acesso

As regras de acesso que envolvem o modificador *default* só fazem sentido se usarmos a estrutura de pacotes - caso as classes sejam declaradas sem serem pertencentes a pacotes, serão consideradas como parte do pacote *default* e campos e métodos declarados sem modificadores serão públicos para todas as outras classes do pacote *default*.

*regras de sobreposição de métodos*

Devemos lembrar também que existem regras especiais para a sobreposição de métodos, dependendo de como estes foram declarados. Em geral, métodos já declarados não podem ser sobrepostos por métodos com declarações mais restritivas. As regras de sobreposição de métodos são:

- Um método declarado como `private` em uma classe ancestral pode ser sobreposto por métodos declarados como `private`, *default*, `protected` ou `public` em uma classe descendente.
- Um método declarado como *default* em uma classe ancestral pode ser sobreposto por métodos declarados como *default*, `protected` ou `public` em uma classe descendente, mas não pode ser sobreposto por um método declarado como `private`.
- Um método declarado como `protected` em uma classe ancestral pode ser sobreposto por métodos declarados como `protected` ou `public` em uma classe descendente, mas não pode ser sobreposto por um método declarado como `private` ou *default*.

- Um método declarado como `public` em uma classe ancestral somente pode ser sobreposto por métodos declarados como `public` em uma classe descendente, não podendo ser sobreposto por um método declarado como `private`, *default* ou `protected`.

## 10.4 Pacotes padrão de Java

*pacotes padrão de Java*

Muito do poder de Java vem de seus pacotes padrão - pacotes que são parte de qualquer distribuição de Java, incluindo o *Java Development Kit* e *Java Runtime Environment*. Java possui pacotes de classes para criação de interfaces gráficas (como aplicações independentes ou *applets* que são executadas por um navegador), entrada e saída usando arquivos e mecanismos semelhantes, criação e manipulação de estruturas de dados, implementação de aplicações em rede, processamento de textos (incluindo internacionalização), representação de valores numéricos com precisão arbitrária e muitas outras.

Algumas das classes incluidas nestes pacotes serão vistas nas seções 2 e 3 deste livro. Classes pertencentes aos pacotes devem ser obrigatoriamente importadas por classes que as queiram usar, através da palavra-chave `import`. Uma exceção são as classes pertencentes ao pacote `java.lang`, que são chamadas as classes-base de Java. Estas classes não precisam ser importadas para que possam ser usadas. Algumas classes deste pacote são `String`, `Math` (que contém métodos estáticos para cálculo de funções matemáticas que serão vistas no capítulo 13), `System` (que contém funções diversas como `currentTimeMillis` e `exit`, e classes para encapsulação de valores nativos (vistas no apêndice A).

## 10.5 Exercícios do capítulo 10

**Exercício 10.1:** ⋆

Explique o que aconterceria se os campos `hora`, `minuto` e `segundo` da classe `Hora` (listagem 10.2) fossem declarados como `protected`. Considere o que aconteceria com as classes `DataHora` e `DemoDataHora` (listagens 10.3 e 10.4).

**Exercício 10.2:** ⋆

O que aconteceria se a linha `import DataHora.*;` da classe `DemoDataHora` (listagem 10.4) fosse escrita como `import DataHora.DataHora;` ?

**Exercício 10.3:** ⋆

Por quê a classe `DemoDataHora` (listagem 10.4) não precisou ser declarada como `public` ?

**Exercício 10.4:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1 /* Esta classe faz parte de todos os pacotes em br.univap.bioinfo */
2 package br.univap.bioinfo.*;
3
4 class ExtremamenteGenerica
5   {
6   private int valor;
7   }
```

**Exercício 10.5:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  package br.univap.bioinfo.geral;
2
3  protected class Coordenada
4    {
5    protected int x,y;
6    }
```

**Exercício 10.6:** ⋆

Explique, com suas palavras, porquê o método `moveRobô` da classe `ArenaBasica` (listagem **??**) não pode simplesmente usar diretamente o campo `posiçãoXAtual` da classe `RoboBasico` (listagem **??**), uma vez que o mesmo foi declarado como `protected` e as duas classes pertencem ao mesmo pacote.

**Exercício 10.7:** ⋆⋆

Identifique e explique o(s) erro(s) nas classes abaixo.

```
1  package br.univap.bioinfo.Geometria;
2
3  public class Ponto2D
4    {
5    protected double x,y;
6    public Ponto2D(double _x,double _y)
7      {
8      x = _x; y = _y;
9      }
10   }
```

```
1  package br.univap.bioinfo.Geometria;
2
3  public class Ponto3D extends Ponto2D
4    {
5    protected double z;
6    public Ponto3D(double _x,double _y,double _z)
7      {
8      x = _x; y = _y; z = _z;
9      }
10   }
```

**Exercício 10.8:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  package demo.publicas;
2
3  public class Ponto3D
4    {
5    default double x,y,z;
6    public Ponto2D(double _x,double _y,double _z)
7      {
8      x = _x; y = _y; z = _z;
9      }
10   }
```

**Exercício 10.9:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  import *;
2
3  class Teste
4    {
5    public static void main(String[] args)
6      {
7      Ponto2D p = new Ponto2D(0,0);
8      Pessoa m = new Pessoa("Cay Horstmann",124123,
9                            new Data((byte)12,(byte)7,(short)1953));
10     }
11   }
```

**Exercício 10.10:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  import *.Data,*.Pessoa;
2
3  class Teste
4    {
5    public static void main(String[] args)
6      {
7      Pessoa m = new Pessoa("Gary Cornell",109834124,
8                            new Data((byte)3,(byte)11,(short)1949));
9      Data hoje = new Data((byte)22,(byte)5,(short)2001);
10     }
11   }
```

# Parte II

# Classes de Java para Manipulação de Dados



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

# Capítulo 11

# Arrays em Java

## 11.1 Introdução

Arrays[1] são estruturas de dados em Java que permitem o armazenamento de várias variáveis de um mesmo tipo ou instâncias de uma mesma classe usando uma única referência e um índice de acesso. Cada um dos valores individuais (ou *elementos*) do array poderá ser acessado individualmente, mas o array inteiro poderá ser processado como uma única entidade caso seja desejado, simplificando bastante algumas tarefas.

*arrays*

*elementos de arrays*

Como um exemplo, considere uma classe que deva encapsular 1440 valores de ponto flutuante, correspondentes a medidas de temperatura obtidas a cada minuto em um dia inteiro em uma estação meteorológica. Claramente usar uma variável para cada medida seria inviável - mesmo somente a declaração e inicialização individual destas variáveis seria extremamente trabalhoso e sujeito a falhas. O problema seria bastante simplificado se somente um nome único fosse dado às variáveis, e índices fossem usados para discriminar entre as diferentes medidas.

*índices*

Similarmente, podemos considerar o problema de representar uma turma de alunos, através de um conjunto de instâncias da classe `Aluno`. Uma classe `Turma` poderia conter um número pré-definido e limitado de referências à instâncias de `Aluno`s, mas novamente a declaração e manipulação de muitas referências independentemente seria complexa e poderia levar a erros.

Os exemplos mostrados acima representam arrays *unidimensionais*, isto é, cujos elementos podem ser acessados por um único índice. Arrays podem ser *multidimensionais*, de forma com que tenhamos que usar mais de um índice para acessar um elemento. Um exemplo simples de array multidimensional é uma matriz matemática, onde valores possam ser acessados especificando-se uma linha e uma coluna, que serão os dois índices usados.

*arrays unidimensionais*

*arrays multidimensionais*

Neste capítulo veremos como arrays de tipos nativos e de instâncias de classes podem ser criados e usados. Veremos também arrays multidimensionais, irregulares, e encapsulamento de arrays em classes para maior flexibilidade e praticidade.

## 11.2 Arrays unidimensionais

A declaração de arrays em Java é feita usando a notação dos colchetes: uma variável declarada

*declaração de arrays*

[1] O autor prefere usar o termo *array* ao invés de *vetor* para evitar a confusão com a classe `Vector`, apresentada no capítulo **??**.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

como sendo de um tipo específico seguido de um par de colchetes será uma referência à um array de elementos daquele tipo. O trecho de código mostrado na listagem 11.1 mostra a declaração de alguns arrays de tipos nativos.

Listagem 11.1: Trecho de código que demonstra a declaração de arrays em Java.

```
1 int[] posiçõesDeMemória;
2 char[] letrasDoAlfabeto;
3 double[] medidasDeTemperatura;
```

No trecho de código mostrado na listagem 11.1, `posiçõesDeMemória` será uma referência a um array de valores do tipo `int`; `letrasDoAlfabeto` será uma referência a um array de valores do tipo `char` e `medidasDeTemperatura` será uma referência a um array de valores do tipo `double`. As declarações de arrays podem ser feitas de maneira ligeiramente diferente, como por exemplo, `int posiçõesDeMemória[];` (com os colchetes depois do nome da referência ao invés de depois do tipo de dado), mas a forma apresentada como exemplo anteriormente é preferida pois declara que a referência será a um array de um determinado tipo.

*alocação de arrays*

Não basta somente declarar as referências a arrays, estas devem ser inicializadas (ter memória alocada para seus elementos). No exemplo mostrado na listagem 11.1, as referências foram simplesmente declaradas, mas não se sabe quantas posições cada array deverá ter. Um array deve ser inicializado com a palavra-chave `new`, seguida do tipo de dado a ser alocado e do número de elementos a alocar, entre colchetes. O trecho de código mostrado na listagem 11.2 demonstra várias maneiras de inicializar de arrays.

Listagem 11.2: Trecho de código que demonstra a declaração e inicialização de arrays em Java.

```
1 int[] posiçõesDeMemória = new int[1024];
2 int quantidadeNecessária = 32768;
3 byte[] vetorNumérico = new byte[quantidadeNecessária];
4 char[] letrasDoAlfabeto = {'a','b','c','d','e','f','g','h','i','j','k','l','m',
5                            'n','o','p','q','r','s','t','u','v','w','x','y','z'};
6 double[] medidasDeTemperatura;
7 medidasDeTemperatura = new double[24*60*60];
8 double[] duplicata = medidasDeTemperatura;
```

A maneira mais comum de inicialização é demonstrada na primeira linha do trecho de código na listagem 11.2, na inicialização do array `posiçõesDeMemória` de valores do tipo `int` - ele é declarado e inicializado na mesma linha. Após a inicialização, podemos acessar seus elementos usando o nome da referência seguida do índice do elemento entre colchetes.

*índice de arrays*

O índice para acesso a um elemento de um array deverá ser um valor entre zero e o tamanho do array menos um. Desta forma `posiçõesDeMemória[0]` corresponderá ao primeiro elemento do array, e `posiçõesDeMemória[1023]` corresponderá ao último elemento. Não é possível acessar elementos fora desta faixa, e o índice deve ser sempre inteiro e positivo. Assim, as declarações `posiçõesDeMemória[-1]`, `posiçõesDeMemória[1024]` e `posiçõesDeMemória[5.4]` são consideradas incorretas.

Um array declarado deve ser inicializado com um tipo compatível com o declarado. Desta forma, a declaração `int[] array = new char[100];` não será válida, e causará um erro de compilação.

Na inicialização do array devemos especificar o seu tamanho entre colchetes. Este tamanho não precisa ser uma constante definida em tempo de compilação: pode ser um valor obtido de qualquer outra fonte, e pode ser determinado em tempo de execução, como mostrado na inicialização

da referência `vetorNumérico`. O tamanho mínimo aceitável para um array é zero: como os índices são contados a partir do zero, um array declarado com tamanho zero terá um único elemento.

Um array, depois de inicializado, não pode ter seu tamanho modificado, embora possamos usar a referência para apontar para outro array. Desta forma as declarações `boolean[] respostas = new boolean[12]; respostas = new boolean[144];` estarão corretas, mas os valores do primeiro array (de 12 posições) serão perdidos quando a referência apontar para o array de 144 posições.

*arrays de valores constantes*

Arrays que deverão representar valores constantes (ou valores definidos em tempo de compilação, que devem ser inicializados ao mesmo tempo e uma só vez e possivelmente não serão modificados posteriormente) podem ser inicializados diretamente com os valores dos elementos do array, que devem ser especificados como uma lista de valores do tipo dos elementos do array, separados por vírgulas e cercados por chaves, como mostrado na declaração da referência `letrasDoAlfabeto` no trecho de código na listagem 11.2. Evidentemente este modo de inicialização só é prático para arrays com um número relativamente pequeno de elementos. Inicializações deste tipo devem ser feitas quando da declaração do array, não é possível declarar o array e inicializá-lo com uma lista de valores posteriormente.

Arrays inicializados com listas de constantes de valores do tipo `float` devem ser declarados de forma que as listas de constantes sejam do tipo correto: a declaração `float[] série = 10,1,0.1,0.01,0.001;` causará um erro de compilação pois o compilador assumirá que os valores decimais são do tipo `double`. Estas constantes devem ser precedidas de um *cast* para o tipo `float` ou ter o sufixo `'f'` que indica que os valores são do tipo `float`.

A declaração de arrays com listas de valores como foi feito para a referência `letrasDoAlfabeto` não torna o array nem seus elementos constantes: posteriormente, o código poderia conter o comando `letrasDoAlfabeto[0] = '?';`, que seria compilado e executado corretamente. O uso do modificador `final` antes da declaração do array fará com que somente o array completo seja final: todos os seus campos poderão ser modificados individualmente, mas não o array como um todo.

Uma outra maneira de inicialização de arrays bastante utilizada em classes é a declaração separada da inicialização, como mostrado no trecho de código na listagem 11.2 para a referência `medidasDeTemperatura`. A declaração pode ser feita em um bloco da classe e a inicialização em outra: esta técnica é bastante usada quando encapsulamos arrays em classes, declarando o array como sendo um campo da classe (podendo ser acessado por todos os métodos da classe) e inicializando-o no construtor (que é o primeiro método a ser chamado, evitando que outros métodos acessem o array não inicializado). Podemos ver na inicialização do array `medidasDeTemperatura` que o tamanho do array não é dado por uma constante, podendo ser o resultado de uma operação qualquer.

*cópia de referências para arrays*

Uma última maneira de inicializarmos um array é criando uma cópia de uma referência a um array já inicializado, como mostrado na última linha do trecho de código na listagem 11.2. Neste caso, o acesso aos elementos do array poderá ser feito por qualquer uma das duas referências (`duplicata` e `medidasDeTemperatura`). Se modificarmos um elemento de uma das referências e recuperarmos o mesmo elemento via a outra referência, o valor recuperado será o modificado. Vale a pena notar que somente podemos fazer cópias de referências quando a referência original já tiver sido inicializada com a palavra-chave `new`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

*valores* default *para elementos de arrays*

Quando arrays de tipos nativos são inicializados, os elementos automaticamente recebem valores *default*: elementos de arrays de valores booleanos serão inicializados com `false`, elementos de arrays de valores numéricos (`char`, `byte`, `short`, `int`, `float`, `long`, `double`) serão inicializados com zeros.

Os elementos de um array inicializado podem ser utilizado como campos ou variáveis independentes. Geralmente um array será populado (isto é, terá seus elementos inicializados) com um laço cuja variável de controle cobre os valores aceitáveis para índices do array, modificando o elemento naquele índice.

A listagem 11.3 demonstra o uso de um array para cálculo da somatória que, se feita com infinitos termos, seria igual a $\pi^2/6$. O cálculo é feito usando um array para armazenar os componentes da somatória.

Listagem 11.3: A classe `CalculoPiQuadradoSobre6`, que calcula o valor de pi quadrado sobre seis.

```
1  /**
2   * A classe CalculoPiQuadradoSobre6, que calcula o valor de pi quadrado sobre seis,
3   * usando uma a série matemática e um array.
4   */
5  class CalculoPiQuadradoSobre6
6    {
7   /**
8    * O método main permite a execução desta classe. Este método declara e inicializa
9    * um array de valores de ponto flutuante, e calcula a somatória de seus elementos
10   * usando um método estático como subrotina.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Consideramos que 1000000 iterações serão calculadas.
17     int iterações = 1000000;
18     // Declaramos um array de valores de ponto flutuante com o tamanho adequado. A
19     // cada elemento do array corresponderá um termo da série a ser calculada.
20     double[] termosDaSérie = new double[iterações];
21     // Calculamos o valor de cada termo da série usando um laço, e armazenamos o
22     // valor do termo da série no elemento correspondente do array
23     for(int contador=0;contador<termosDaSérie.length;contador++)
24       {
25       double denominador = contador+1.;
26       termosDaSérie[contador] = 1./(denominador*denominador);
27       }
28     // Com os termos calculados, vamos ver o resultado da somatória da série
29     // com diferentes quantidades de termos. Para comparação, o valor de pi
30     // ao quadrado sobre seis, calculado com uma calculadora, é 1.64493406684823
31     calculaEMostraSomatória(termosDaSérie,10);              // 1.5497677311665408
32     calculaEMostraSomatória(termosDaSérie,100);             // 1.6349839001848923
33     calculaEMostraSomatória(termosDaSérie,1000);            // 1.6439345666815615
34     calculaEMostraSomatória(termosDaSérie,10000);           // 1.6448340718480652
35     calculaEMostraSomatória(termosDaSérie,100000);          // 1.6449240668982423
36     calculaEMostraSomatória(termosDaSérie,1000000);         // 1.64493306684877
37     }
38
39  /**
40   * O método calculaEMostraSomatória calcula e mostra a somatória de um array de
41   * valores de ponto flutuante passado como argumento, usando um índice máximo também
42   * passado como argumento.
43   */
44   private static void calculaEMostraSomatória(double[] array,int índiceMáximo)
45     {
46     double somatória = 0;
47     for(int índice=0;índice<índiceMáximo;índice++)
```

```
48      {
49      somatória += array[índice];
50      }
51    System.out.println("Índice Máximo:"+índiceMáximo+" Somatória:"+somatória);
52    }
53
54  } // fim da classe CalculoPiQuadradoSobre6
```

Alguns pontos interessantes da listagem 11.3 são:

- O array é inicializado usando como tamanho o valor de uma variável do programa, desta forma caso seja desejável modificar o tamanho do array, basta modificar a variável `iterações`.
- O laço que inicializa os termos da somatória será executado enquanto sua variável de controle (`contador`) for menor do que o tamanho do array. Usamos, para este controle, o campo `length` que pertence a cada array unidimensional, e que indica o número de elementos no array. O valor de `termosDaSérie.length` é igual a `iterações`. Note que o valor é um campo do array, e não um método: a chamada a `termosDaSérie.length()` estaria incorreta. O campo `length` comporta-se como um campo final, não podendo ser modificado.
- O array pode ser passado como argumento a um método. Se isto for feito, todos os elementos do array estarão disponíveis para acesso e uso dentro do corpo do método. Note que o primeiro argumento para o método `calculaEMostraSomatória` é do tipo array unidimensional de `double`s.

Um segundo exemplo de uso de arrays, mais interessante e mais complexo, é mostrado na classe `ArrayDeFloats` (listagem 11.4). Esta classe encapsula um array de valores do tipo `float` cujo tamanho é definido pelo programador usuário da classe. A classe permite a interação do usuário com o array somente através de métodos que verificam os índices sendo usados, evitando que índices inválidos sejam utilizados.

Listagem 11.4: A classe `ArrayDeFloats`, que encapsula um array de valores do tipo `float`.

```
/**
 * A classe ArrayDeFloats, que encapsula um array de valores de ponto flutuante, com
 * vários métodos que a tornam mais útil e versátil do que um array comum.
 */
class ArrayDeFloats // declaração da classe
  {
 /**
  * Declaração dos campos da classe
  */
  private float[] array; // o array encapsulado

 /**
  * O construtor para a classe ArrayDeFloats, que recebe um argumento que será o
  * número de posições (ou valores do tipo float) que a instância da classe será
  * capaz de armazenar.
  * @param número o número de posições para valores do tipo float a serem alocadas
  */
  ArrayDeFloats(int número)
    {
    array = new float[número]; // alocamos memória para o array encapsulado
    }

 /**
  * O método tamanho retorna o tamanho do array encapsulado. Uma vez que o campo que
  * representa o array é declarado como sendo privado, é uma boa idéia ter uma maneira
  * de acessar o tamanho do array de fora desta classe.
```

```
27    * @return o número de posições para valores do tipo float no array encapsulado
28    */
29    public int tamanho()
30      {
31      return array.length;
32      }
33
34   /**
35    * O método modifica recebe como argumentos uma posição e um valor, modificando no
36    * array encapsulado o valor na posição especificada. O valor somente será modificado
37    * se a posição for válida, isto é, estiver entre zero e o tamanho máximo do array
38    * menos um. Caso uma posição não válida seja passada, o método não executará nenhum
39    * comando.
40    * @param posição a posição do array que será modificada
41    * @param valor o valor que será armazenado na posição especificada
42    */
43    public void modifica(int posição,float valor)
44      {
45      if ((posição >= 0) && (posição < array.length))
46        array[posição] = valor;
47      }
48
49   /**
50    * O método valor retorna o valor armazenado em uma certa posição, que é passada como
51    * argumento para o método. O valor armazenado somente será retornado se a posição
52    * for válida, isto é, estiver entre zero e o tamanho máximo do array menos um. Caso
53    * uma posição não válida seja passada, o método retornará a constante Float.NaN, que
54    * representa um valor de ponto flutuante que não é um número. Em exemplos e
55    * exercícios posteriores veremos maneiras mais elegantes de tratar esses tipos de
56    * problema.
57    * @param posição a posição do array cujo valor será retornado
58    * @return o valor que estava armazenado na posição especificada
59    */
60    public float valor(int posição)
61      {
62      if ((posição >= 0) && (posição < array.length))
63        return array[posição];
64      else return Float.NaN;
65      }
66
67   /**
68    * O método paraArray retorna o array encapsulado como um array de floats. Este
69    * método pode ser útil caso seja necessário usar o array "desencapsulado", para,
70    * por exemplo, passá-lo como argumento para um método que somente aceita arrays.
71    * @return o array encapsulado por esta classe
72    */
73    public float[] paraArray()
74      {
75      return array;
76      }
77
78   /**
79    * O método menorValor retorna o menor valor presente no array encapsulado. Este
80    * método usa um algoritmo simples que considera que no seu primeiro passo o menor
81    * valor encontrado é o primeiro valor do array, e varre os valores restantes,
82    * modificando o menor valor armazenado se um elemento do array for menor que o
83    * armazenado.
84    * @return o menor valor presente no array encapsulado
85    */
86    public float menorValor()
87      {
88      float menorAtéAgora = array[0];
89      for(int c=1;c<array.length;c++)
90        if (array[c] < menorAtéAgora)
91          menorAtéAgora = array[c];
92      return menorAtéAgora;
93      }
94
95   /**
```

```
96    * O método maiorValor retorna o maior valor presente no array encapsulado. Este
97    * método usa um algoritmo simples que considera que no seu primeiro passo o maior
98    * valor encontrado é o primeiro valor do array, e varre os valores restantes,
99    * modificando o maior valor armazenado se um elemento do array for maior que o
100   * armazenado.
101   * @return o maior valor presente no array encapsulado
102   */
103   public float maiorValor()
104     {
105     float maiorAtéAgora = array[0];
106     for(int c=1;c<array.length;c++)
107       if (array[c] > maiorAtéAgora)
108         maiorAtéAgora = array[c];
109     return maiorAtéAgora;
110     }
111
112  /**
113   * O método toString retorna os valores do array encapsulado formatados em uma
114   * String.
115   * @return uma String contendo uma representação dos valores do array encapsulado.
116   */
117   public String toString()
118     {
119     String resultado = "O array tem "+array.length+" elementos:\n";
120     for(int c=0;c<array.length;c++)
121       resultado += array[c]+" ";
122     return resultado;
123     }
124
125  } // fim da classe ArrayDeFloats
```

Os pontos de interesse da classe `ArrayDeFloats` (listagem 11.4) são:

- A classe encapsula um array de floats, fornecendo métodos para recuperar e modificar o array. O campo que representa o array é declarado na classe, fora dos métodos, para que possa ser acessado de qualquer método. O array é inicializado (memória alocada) no construtor da classe, desta forma quando métodos da classe forem executados saberemos que o array já foi inicializado, uma vez que o construtor será o primeiro método a ser chamado.
- Como o array foi declarado como sendo `private` da classe, o campo `length` do array também o será. Como é útil sabermos o tamanho do array, criamos o método `tamanho` para retornar este valor.
- Dois métodos permitem a modificação e recuperação de valores do array encapsulado: `modifica` e `valor`. Ambos os métodos tem um mecanismo primitivo para evitar que índices incorretos sejam usados para acessar os elementos do array encapsulado.
- A classe fornece um método `paraArray` que retorna o array encapsulado - este método pode ser útil caso seja necessário montar um array a partir dos valores encapsulados, evitando múltiplas chamadas ao método `valor`, o que é computacionalmente custoso. O tipo de retorno deste método é `float[]`, ou seja, array de floats.
- Dois métodos adicionais (`maiorValor` e `menorValor`) demonstram as vantagens do encapsulamento de um array ao invés do uso direto: podemos incluir métodos que calculam, inicializam ou fazem operações de interesse no array, criando métodos que façam o processamento.

A classe `DemoArrayDeFloats`, mostrada na listagem 11.5, demonstra o uso de uma instância da classe `ArrayDeFloats`.

Listagem 11.5: A classe `DemoArrayDeFloats`, que demonstra o uso de instâncias da classe `ArrayDeFloats`.

```
1  /**
2   * A classe DemoArrayDeFloats, que demonstra o uso de instâncias da classe
3   * ArrayDeFloats.
4   */
5  class DemoArrayDeFloats
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * uma instância da classe ArrayDeFloats, e demonstra chamada de seus métodos.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     // Declara e inicializa a instância da classe
16     ArrayDeFloats af = new ArrayDeFloats(25);
17     // Vamos popular o array com valores obtidos com cálculos simples e um laço.
18     for(int i=-3;i<40;i++)        // os valores do laço não correspondem diretamente
19       af.modifica(i,(float)1./i); // aos índices aceitáveis para o array encapsulado,
20                                   // porém somente os índices com valores entre 0 e 24
21                                   // serão realmente usados.
22     System.out.println(af); // imprimimos uma String contendo array encapsulado.
23     // Imprimimos o maior e o menor valor do array encapsulado
24     System.out.println("Maior valor:"+af.maiorValor());
25     System.out.println("Menor valor:"+af.menorValor());
26     // Imprimimos os valores em determinadas posições do array encapsulado
27     System.out.println("Na posição  0:"+af.valor(0));
28     System.out.println("Na posição 24:"+af.valor(24));
29     System.out.println("Na posição 25:"+af.valor(25));
30     // Imprimimos os valores em determinadas posições do array encapsulado, mas usando
31     // a representação desencapsulada
32     float[] cópia = af.paraArray();
33     System.out.println("Na posição  0:"+cópia[0]); // indiretamente
34     System.out.println("Na posição  0:"+af.paraArray()[0]); // diretamente
35     }
36
37   } // fim da classe DemoArrayDeFloats
```

Na classe `DemoArrayDeFloats` (listagem 11.5) podemos ver a declaração de uma instância da classe `ArrayDeFloats` e chamadas a métodos desta classe. Mesmo com chamadas com argumentos nitidamente incorretos (tentativa de acesso a elementos fora da faixa dos valores válidos), a execução da classe não apresentará erros, pois os métodos de acesso foram preparados para ignorar tentativas de acesso inválido.

A classe `DemoArrayDeFloats` apresenta dois pontos interessantes que estão relacionados ao método `paraArray`. Uma chamada ao método é feita, e seu resultado armazenado na referência `cópia`, que não foi inicializada, somente tendo sido declarada. Esta chamada ao método faz com que o array encapsulado na instância `af` e o array `cópia` apontem para a mesma posição de memória, desta forma, qualquer modificação feita usando a referência `cópia` alterará o array encapsulado. Se este não é o comportamento desejado, podemos reescrever o método `paraArray` para que este retorne uma cópia ou clone do array encapsulado.

O segundo ponto interessante é a sintaxe da segunda chamada ao método `paraArray`, mostrada como `af.paraArray()[0]`, que permite que acessemos um elemento de um resultado de chamada a método. Embora a sintaxe seja incomum, está absolutamente correta: o resultado da operação `af.paraArray()` é um array de floats, e podemos acessar seu primeiro elemento com o sufixo `[0]`.

### 11.2.1 Arrays de instâncias de classes

*arrays de instâncias de classes*

Arrays de instâncias de classes (tanto já existentes em Java quanto definidas pelo programador) podem ser declaradas de forma bem similar à arrays de valores de tipos nativos. A diferença principal é que a inicialização do array deve ser seguida da inicialização dos elementos do array, que deve ser feita da mesma maneira que fazemos com instâncias de classes comuns - geralmente, através da palavra-chave `new` e da chamada ao construtor da classe.

*polimorfismo e arrays*

Uma diferença fundamental entre os arrays de tipos nativos e de instâncias de classes é que elementos de arrays nativos devem ser sempre do mesmo tipo, enquanto que elementos de arrays de instâncias podem conter instâncias de qualquer classe que seja herdeira da classe usada para declaração do array (aplicação de polimorfismo em arrays de instâncias de classes). Desta forma, se declaramos um array de instâncias da classe `Pessoa`, por exemplo, poderíamos inicializar um dos elementos deste array como sendo uma instância da classe `Aluno`, se a classe `Aluno` for herdeira, direta ou indiretamente, da classe `Pessoa`.

Um exemplo é dado pela classe `Equipe`, mostrada na listagem 11.6. Esta classe cria e inicializa um array de instâncias da classe `Funcionario` (mostrada na listagem 8.8).

Listagem 11.6: A classe `Equipe`, que demonstra o uso de um array de instâncias da classe `Funcionario`.

```
1  /**
2   * A classe Equipe, que demonstra o uso de um array de instâncias da classe
3   * Funcionario. O método main desta classe declara e inicializa o array de instâncias.
4   */
5  class Equipe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método declara e inicializa o
9    * array de instâncias da classe Funcionario, demonstrando características e
10   * problemas potenciais de arrays de instâncias de classes.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Declaramos um array de funcionários com cinco posições
17     Funcionario[] equipe = new Funcionario[5];
18     // Temos agora cinco posições para armazenar instâncias da classe Funcionario no
19     // array, mas as instâncias ainda não foram alocadas ! Devemos fazer isto
20     // manualmente para cada elemento do array.
21     equipe[0] = new Funcionario("Alan Moore",234924,
22                                 new Data((byte)12,(byte)10,(short)1955),
23                                 new Data((byte)22,(byte)3,(short)1978),
24                                 12500f);
25     equipe[1] = new Funcionario("Charles Vess",987120,
26                                 new Data((byte)8,(byte)9,(short)1932),
27                                 new Data((byte)5,(byte)1,(short)1943),
28                                 43320f);
29     equipe[2] = new Funcionario("Dave McKean",876451,
30                                 new Data((byte)27,(byte)12,(short)1940),
31                                 new Data((byte)17,(byte)11,(short)1968),
32                                 38200f);
33     equipe[3] = new ChefeDeDepartamento("Neil Gaiman",769721,
34                                 new Data((byte)15,(byte)9,(short)1959),
35                                 new Data((byte)18,(byte)8,(short)1980),
36                                 41000f,
37                                  "Departamento de Criação",
38                                 new Data((byte)30,(byte)4,(short)1987));
39     // Podemos imprimir diretamente os elementos do array pois são instâncias da classe
40     // Funcionario, que contém o método mágico toString.
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
41      System.out.println(equipe[0]);
42      System.out.println(equipe[3]);
43      System.out.println(equipe[4]); // imprimirá null !
44      }
45
46    } // fim da classe Equipe
```

Os principais pontos da classe `Equipe` são:

- O array de instâncias da classe `Funcionario` é declarado e inicializado de maneira similar aos arrays de valores de tipos nativos. Após a inicialização, o array também será preenchido com um valor constante. No caso de arrays de instâncias de classes, todos os elementos do array receberão o valor `null`.
- Cada elemento do array é usado como se fosse uma referência individual à uma instância da classe `Funcionario`, e deve ser inicializado individualmente através da palavra-chave `new` e do construtor da classe `Funcionario`.
- Podemos inicializar um dos elementos do array de instâncias de `Funcionario` com uma instância da classe `ChefeDeDepartamento`, uma vez que a classe `ChefeDeDepartamento` é herdeira da classe `Funcionario`.
- Cada elemento do array segue a mesma regra das referências à instâncias de classes: se o comando `equipe[4] = equipe[2]` fosse executado, as duas referências estariam apontando para a mesma instância, e qualquer modificação feita usando uma das duas referências estaria afetando a mesma instância.

Um exemplo mais detalhado é dado pela classe `ArrayDeObjetosGeometricos`, mostrada na listagem 11.7. Esta classe encapsula um array de referências à instância `ObjetoGeometrico` (listagem 9.5).

Listagem 11.7: A classe `ArrayDeObjetosGeometricos`, que encapsula um array de instâncias de qualquer classe que implemente a interface `ObjetosGeometricos`.

```
1  /**
2   * A classe ArrayDeObjetosGeometricos, que encapsula um array de instâncias de qualquer
3   * classe que implemente a interface ObjetosGeometricos.
4   */
5  class ArrayDeObjetosGeometricos // declaração da classe
6    {
7   /**
8    * Declaração dos campos da classe
9    */
10   private ObjetoGeometrico[] array; // o array encapsulado
11
12  /**
13   * O construtor para a classe ArrayDeObjetosGeometricos, que recebe um argumento
14   * que será o número de posições que a instância da classe será capaz de armazenar.
15   * @param número o número de posições para armazenamento de instâncias de classes
16   *        que implementem a interface ObjetoGeometrico
17   */
18   ArrayDeObjetosGeometricos(int número)
19     {
20     array = new ObjetoGeometrico[número]; // alocamos memória para o array encapsulado
21     // Sabemos que cada uma das posições do array apontará para null, então não
22     // precisamos inicializar o array.
23     }
24
25  /**
26   * O método tamanho retorna o tamanho do array encapsulado. Uma vez que o campo que
27   * representa o array é declarado como sendo privado, é uma boa idéia ter uma maneira
28   * de acessar o tamanho do array de fora desta classe.
29   * @return o número de posições no array encapsulado
```

```
30    */
31    public int tamanho()
32      {
33      return array.length;
34      }
35
36   /**
37    * O método modifica recebe como argumentos uma posição e uma instância de uma classe
38    * que implementa a instância ObjetoGeometrico, modificando no array encapsulado a
39    * instância na posição especificada. A instância somente será modificada se a posição
40    * for válida, isto é, estiver entre zero e o tamanho máximo do array menos um.
41    * Caso uma posição não válida seja passada, o método não executará nenhum comando.
42    * Note que não dizemos qual é a classe cuja instância será passada como argumento,
43    * o que seria restritivo demais. Em vez disto, consideramos que o argumento será
44    * uma instância de qualquer classe que implementa a interface ObjetoGeometrico,
45    * @param posição a posição do array que será modificada
46    * @param umObjeto uma instância de uma classe que implementa a interface
47    *        ObjetoGeometrico
48    */
49    public void modifica(int posição,ObjetoGeometrico umObjeto)
50      {
51      if ((posição >= 0) && (posição < array.length))
52        array[posição] = umObjeto;
53      }
54
55   /**
56    * O método qualObjeto retorna o objeto geométrico armazenado em uma certa posição,
57    * que é passada como argumento para o método. O objeto geométrico armazenado somente
58    * será retornado se a posição for válida, isto é, estiver entre zero e o tamanho
59    * máximo do array menos um. Caso uma posição não válida seja passada, o método
60    * retornará null, significando objeto não-válido. Em exemplos e exercícios
61    * posteriores veremos maneiras mais elegantes de tratar esses tipos de problema.
62    * @param posição a posição do array. Se for válida, o objeto armazenado nesta
63    *        posição será retornado
64    * @return o objeto geométrico que estava armazenado na posição especificada
65    */
66    public ObjetoGeometrico qualObjeto(int posição)
67      {
68      if ((posição >= 0) && (posição < array.length))
69        return array[posição];
70      else return null;
71      }
72
73   /**
74    * O método toString retorna os valores do array encapsulado formatados em uma
75    * String.
76    * @return uma String contendo uma representação dos valores do array encapsulado.
77    */
78    public String toString()
79      {
80      String resultado = "O array tem "+array.length+" elementos:\n";
81      for(int c=0;c<array.length;c++)
82        resultado += " "+(c+1)+":"+
83                     array[c]+"\n"; // implicitamente chamamos o método toString
84      return resultado;               // da classe que implementa o objeto geométrico
85      }
86
87    } // fim da classe ArrayDeObjetosGeometricos
```

Alguns pontos interessantes da classe `ArrayDeObjetosGeometricos` são:

- A classe tem uma estrutura semelhante à da classe `ArrayDeFloats` (listagem 11.4): um array declarado como sendo `private`, acessível por todos os métodos da classe, inicializado no construtor; métodos para modificar e retornar elementos do array que verificam os índices passados, um método `toString` que retorna a representação do array como uma String.

- O array encapsulado é de elementos da interface `ObjetoGeometrico`, assim como o segundo argumento para o método `modifica` e o tipo de retorno do método `qualObjeto`. Evidentemente não podemos instanciar a interface `ObjetoGeometrico`, mas com as declarações deste tipo podemos armazenar no array encapsulado, passar e receber como argumentos qualquer instância de qualquer classe que implemente a interface `ObjetoGeometrico`.

A classe `DemoArrayDeObjetosGeometricos`, mostrada na listagem 11.8, demonstra o uso de uma instância da classe `ArrayDeObjetosGeometricos`.

Listagem 11.8: A classe `DemoArrayDeObjetosGeometricos`, que demonstra o uso de instâncias da classe `ArrayDeObjetosGeometricos`.

```
1  /**
2   * A classe DemoArrayDeObjetosGeometricos, que demonstra o uso de instâncias da classe
3   * ArrayDeObjetosGeometricos.
4   */
5  class DemoArrayDeObjetosGeometricos
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações de
9    * uma instância da classe ArrayDeObjetosGeometricos, e demonstra chamada de seus
10   * métodos.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     // Declara e inicializa a instância da classe: um array de objetos geométricos com
17     // capacidade para armazenar oito objetos
18     ArrayDeObjetosGeometricos adog = new ArrayDeObjetosGeometricos(8);
19     // Criamos algumas instâncias com referências de classes que implementam a
20     // interface ObjetosGeometricos
21     Circulo c1 = new Circulo(new Ponto2D(0,0),25.0); // centrado na origem, raio 25
22     Circulo c2 = new Circulo(new Ponto2D(-9,-3),0.1); // centrado em (-9,-3), raio 0.1
23     Retangulo r1 = new Retangulo(new Ponto2D(-1,-1), // um canto está em (-1,-1)
24                                  new Ponto2D(1,1));  // e o outro em (1,1)
25     Retangulo r2 = new Retangulo(new Ponto2D(0,0),  // um canto está em (0,0)
26                                  new Ponto2D(0,0)); // e o outro também !
27     // Adicionamos estas instâncias à instância da classe ArrayDeObjetosGeometricos
28     adog.modifica(0,c1);
29     adog.modifica(1,c2);
30     adog.modifica(3,r1);
31     adog.modifica(120,r2); // este não deverá ser adicionado
32     // imprimimos uma String que representa o array encapsulado.
33     System.out.println(adog);
34     // Imprimimos os objetos em determinadas posições do array encapsulado - alguns
35     // não estão em posições válidas
36     System.out.println("Na posição  0:"+adog.qualObjeto(0));
37     System.out.println("Na posição  3:"+adog.qualObjeto(3));
38     System.out.println("Na posição  6:"+adog.qualObjeto(6));
39     System.out.println("Na posição 99:"+adog.qualObjeto(99));
40     // Qual é a origem do objeto na posição 1 ?
41     System.out.println("Centro do objeto na posição 1 é "+
42                        adog.qualObjeto(1).centro());
43     // Qual é a área do objeto na posição 3 ?
44     System.out.println("Área do objeto na posição 3 é "+
45                        adog.qualObjeto(3).calculaÁrea());
46     // Qual é o perímetro do objeto na posição 123 ? Como tal índice não existe, o
47     // método qualObjeto retornará null, e a máquina virtual Java mostrará um erro
48     // em tempo de execução ao executar a linha a seguir (não podemos executar
49     // métodos usando null como referência !)
50     System.out.println("Perímetro do objeto na posição 123 é "+
51                        adog.qualObjeto(123).calculaPerímetro());
52     } // fim do método main
53
54   } // fim da classe DemoArrayDeObjetosGeometricos
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Os pontos de interesse na classe `DemoArrayDeObjetosGeometricos` são:

- Uma instância da classe `ArrayDeObjetosGeometricos` é criada, assim como várias instâncias de classes que implementam a interface `ObjetoGeometrico` são declaradas e inicializadas como referências independentes.
- Para popular o array encapsulado na classe `ArrayDeObjetosGeometricos` o método `main` chama algumas vezes o método `modifica` daquela classe, para inserir as instâncias criadas no array encapsulado. O método `modifica` restringe o acesso ao o array, impedindo que índices não-válidos sejam usados para acessar o array encapsulado na classe `ArrayDeObjetosGeometricos`.
- Os elementos do array podem ser acessados usando o método `qualObjeto` da classe `ArrayDeObjetosGeometricos`. Este método retornará uma referência a uma classe que herda da classe `ObjetoGeometrico`, e poderemos executar métodos destas usando diretamente o retorno do método `qualObjeto`. A listagem mostra alguns exemplos de chamadas indiretas ao método `toString`, assim como chamadas explícitas aos métodos `centro` e `calculaÁrea`. Se o argumento passado para o método `qualObjeto` for inválido (sendo fora da faixa de valores aceitáveis como índices para o array encapsulado), o método retornará `null`, e neste caso, não poderemos chamar métodos usando este resultado, pois erros em tempo de execução ocorrerão.
- É imporante compreender que o array de objetos geométricos encapsulado dentro da classe `ArrayDeObjetosGeometricos` contém somente cópias às referências criadas na classe `DemoArrayDeObjetosGeometricos`, e que foram passadas para o array encapsulado pelo método `modifica` - o array não contém cópias das instâncias, significando que se pudéssemos modificar as instâncias *depois* de tê-las passado para o array encapsulado, a modificação se refletiria no array.
  Por exemplo, consideremos a instância da classe `Circulo` cuja referência é `c2`. A instância é criada e posteriormente incluída no array encapsulado representado pela instância `adog`. Se esta instância da classe `Circulo` tiver seu raio modificado para 0.0, o segundo elemento do array encapsulado também será modificado, mesmo que a modificação tenha sido feita depois da inserção.

## 11.3 Arrays multidimensionais

*arrays multidimensionais*

Os arrays mostrados nos diversos exemplos até agora são arrays de uma única dimensão ou unidimensionais: para acessar os elementos destes arrays basta usar um único índice. Podemos facilmente criar arrays multidimensionais em Java, onde a posição de cada elemento dentro do array será indicada por dois ou mais índices.

Um exemplo comum de arrays multidimensionais são matrizes matemáticas, que representam valores tabulados em linhas e colunas. A classe `MatrizDeDoubles`, mostrada na listagem 11.9, exemplifica o encapsulamento de um array multidimensional. A classe foi feita propositalmente simples, para uso em vários exercícios, com poucos métodos implementados.

Listagem 11.9: A classe `MatrizDeDoubles`, que encapsula um array bidimensional (matriz) de valores do tipo `double`.

```
1 /**
2  * A classe MatrizDeDoubles, que encapsula uma matriz bidimensional (matriz) de
3  * valores do tipo double.
4  */
5 class MatrizDeDoubles // declaração da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
6   {
7  /**
8   * Declaração dos campos da classe
9   */
10  private int linhas;  // o número de linhas da matriz
11  private int colunas; // o número de colunas da matriz
12  private double[][] matriz; // o array bidimensional (matriz) encapsulado
13
14 /**
15  * O construtor da classe MatrizDeDoubles, que receberá as dimensões da matriz como
16  * argumentos, alocará a matriz e inicializará os valores da matriz para zero.
17  * @param colunas O número de colunas na matriz
18  * @param linhas O número de linhas na matriz
19  */
20  MatrizDeDoubles(int colunas,int linhas)
21    {
22    // Inicializa as variáveis locais da classe com as passadas como argumentos
23    this.colunas = colunas; this.linhas = linhas;
24    // Aloca memória para a matriz
25    matriz = new double[linhas][colunas];
26    // Preenche a matriz com zeros
27    preencheMatriz(0);
28    } // fim do construtor
29
30 /**
31  * O método preencheMatriz, que preenche a matriz encapsulada pela classe com um
32  * valor passado como argumento.
33  * @param valor o valor a ser usado para preencher a matriz.
34  */
35  public void preencheMatriz(double valor)
36    {
37    // Cada elemento da matriz...
38    for(int lin=0;lin<linhas;lin++)
39      for(int col=0;col<colunas;col++)
40        matriz[lin][col] = valor; // ... receberá o valor constante
41    } // fim do método preencheMatriz
42
43 /**
44  * O método maiorValor retorna o maior elemento presente na matriz encapsulada. Este
45  * método usa um algoritmo simples que considera que no seu primeiro passo o maior
46  * valor encontrado é o primeiro valor do array, e varre os valores restantes,
47  * modificando o maior valor armazenado se um elemento do array for maior que o
48  * armazenado.
49  * @return o maior elemento da matriz
50  */
51  public double maiorValor()
52    {
53    // Considera o primeiro valor como maior até agora
54    double maiorAtéAgora = matriz[0][0];
55    // Para cada elemento da matriz...
56    for(int lin=0;lin<linhas;lin++)
57      for(int col=0;col<colunas;col++)
58        if (matriz[lin][col] > maiorAtéAgora) // se for maior que o maior até agora
59          maiorAtéAgora = matriz[lin][col];   // armazena o novo maior valor
60    return maiorAtéAgora;
61    } // fim do método maior
62
63 } // fim da classe MatrizDeDoubles
```

Os pontos interessantes da classe `MatrizDeDoubles` são:

- A classe encapsula uma matriz de valores de ponto flutuante e dois campos para representar o número de linhas e colunas desta matriz. Uma matriz (ou array bidimensional) é declarado de forma similar a um array unidimensional, exceto que dois pares de colchetes são usados para declarar o tipo de dado. A cada par de colchetes corresponderá uma das dimensões da matriz.

- O construtor da classe recebe como argumento os tamanhos das duas dimensões da matriz, e os usa para inicializar a matriz encapsulada. A inicialização (alocação) da matriz é similar à de um array unidimensional, mas dois pares de colchetes, cada um contendo um tamanho de dimensão do array, devem ser especificados.
- O método `preencheMatriz` preenche a matriz com uma constante qualquer. A sua chamada a partir do construtor é redundante, uma vez que a matriz, quando for criada, será preenchida automaticamente com zeros. O método pode ser usado para criar matrizes preenchidas com outros valores. O preenchimento é feito acessando os elementos da matriz, um a um, através de seus dois índices, que são duas variáveis de controle de dois laços.
- O método `maiorValor` funciona de forma similar à do método de mesmo nome da classe `ArrayDeFloats`, procura o maior elemento da matriz comparando um valor inicial com todos os elementos, trocando se for necessário, e retornando o maior valor encontrado ao fim do método.

Um outro exemplo de uso de arrays multidimensionais é dado pela classe `JogoDeDamas`, mostrada na listagem 11.10. Esta classe encapsula um tabuleiro de jogo de damas usando caracteres, com casas vazias sendo representadas pelo caracter `'.'` e casas com peças de jogadores sendo representadas pelos caracteres `'o'` e `'x'`.

Listagem 11.10: A classe `JogoDeDamas`, que encapsula um tabuleiro de jogo de damas.

```
1  /**
2   * A classe JogoDeDamas, que encapsula um tabuleiro de jogo de damas usando um
3   * array de duas dimensões. Casas vazias no tabuleiro são representadas pelo
4   * caracter '.', e casas com peças pelos caracteres 'o' e 'x'. Esta versão da
5   * classe não prevê a representação de peças duplas (damas), que poderiam ser
6   * representadas por '@' e '*'.
7   */
8  class JogoDeDamas // declaração da classe
9    {
10  /*
11   * O tabuleiro é representado por um array de caracteres de duas dimensões, com o
12   * tamanho 8x8. Para evitar a manipulação direta do tabuleiro, o campo que o
13   * representa é declarado como private.
14   */
15   private char tabuleiro[][];
16
17  /**
18   * O construtor da classe, que não recebe nenhum argumento. O construtor alocará
19   * o array de duas dimensões que representa o tabuleiro e inicializará as posições
20   * do tabuleiro de forma a representar o jogo em seu início.
21   */
22   JogoDeDamas()
23     {
24     // A alocação é simples, o tabuleiro tem sempre 8x8 posições
25     tabuleiro = new char[8][8];
26     // Inicializamos todas as posições do tabuleiro com o caracter ponto ('.')
27     // que representa a ausência de peças na posição.
28     for(int linha=0;linha<8;linha++)
29       for(int coluna=0;coluna<8;coluna++)
30         tabuleiro[linha][coluna] = '.'; // todas as posições inicializadas com pontos
31     // Inicialmente, todas as peças estão em posições pré-definidas: as do jogador
32     // que está de frente para o tabuleiro são marcadas com o caracter 'o' ...
33     for(int linha=0;linha<=1;linha++) // somente as duas primeiras linhas
34       for(int coluna=0;coluna<8;coluna++)
35         tabuleiro[linha][coluna] = 'o';
36     // ... e as do outro jogador (que está do lado oposto do tabuleiro) são marcadas
37     // com o caracter 'x'.
38     for(int linha=6;linha<=7;linha++) // somente as duas últimas linhas
39       for(int coluna=0;coluna<8;coluna++)
40         tabuleiro[linha][coluna] = 'x';
41     } // fim do construtor
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
42
43  /**
44   * O método toString não recebe argumentos, e retorna uma String contendo os valores
45   * dos campos da classe formatados. No caso desta classe, o array de duas dimensões
46   * será formatado como se fosse um pequeno tabuleiro de damas.
47   * @return uma String contendo uma representação do tabuleiro de damas
48   */
49   public String toString()
50     {
51     String resultado = ""; // esta String receberá o resultado
52     for(int linha=0;linha<8;linha++) // para todas as linhas do tabuleiro...
53       {
54       for(int coluna=0;coluna<8;coluna++) // para cada coluna do tabuleiro
55         resultado += tabuleiro[linha][coluna]; // concatena a peça naquela posição
56       // e inclui uma quebra de linha no final de cada linha do tabuleiro
57       resultado += "\n";
58       }
59     return resultado;
60     } // fim do método toString
61
62   } // fim da classe JogoDeDamas
```

Os pontos interessantes da classe `JogoDeDamas` são seu construtor, que não somente aloca memória para o array bidimensional que representa o tabuleiro do jogo de damas, mas também inicializa os elementos do array para que o tabuleiro fique na sua configuração inicial, e o método `toString`, que retorna uma String contendo a representação gráfica do tabuleiro.

Arrays multidimensionais podem ter bem mais que duas dimensões. Algumas aplicações matemáticas e de simulações requerem o uso de arrays com mais de três dimensões. A criação destes arrays segue o mesmo padrão dos arrays uni- e bidimensionais - a declaração do array deve ser seguida de um número de pares de colchetes igual ao número de dimensões desejadas, e o acesso aos elementos do array deve ser feito usando um índice para cada dimensão.

### 11.3.1 Arrays irregulares

*arrays irregulares*

Uma característica interessante da maneira em que arrays multidimensionais são implementados em Java é que os arrays não precisam ter o mesmo número de valores para cada dimensão, ou seja, é possível declararmos arrays que sejam irregulares. Por exemplo, um array irregular bidimensional poderia ter cinco linhas, e cada linha pode ter um número diferente de colunas - não é obrigatório que cada linha tenha o mesmo número de colunas.

*arrays de arrays*

Esta característica deve-se ao fato que Java trata arrays multidimensionais como *arrays de arrays*: desta forma um array bidimensional pode ser considerado um array unidimensional onde cada elemento é outro array unidimensional que contém valores de determinado tipo. Um exemplo ilustrativo de array irregular é mostrado na figura 11.1.

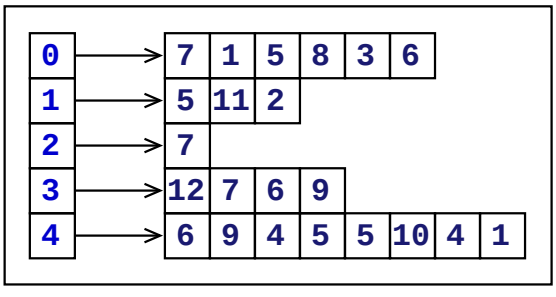



Figura 11.1: Exemplo de array irregular de duas dimensões.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

A coluna à esquerda da figura 11.1 representa o índice da primeira dimensão do array - cada posição na primeira dimensão aponta para um array de valores inteiros.

Um array irregular pode ser declarado da mesma forma que um array multidimensional regular, exceto que somente o tamanho da primeira das dimensões precisa ser especificado - as outras dimensões, podendo ser diferentes, devem ser alocadas em passos subsequentes do método.

Um exemplo relativamente simples de aplicação de arrays irregulares é o triângulo de Pascal. O triângulo de Pascal é uma série de valores onde cada elemento pode ser calculado como a soma do elemento acima com o elemento acima e à esquerda. O primeiro e último elementos de cada linha do triângulo são sempre iguais a um. O triângulo de Pascal de seis linhas é:

```
1  1
1  2   1
1  3   3   1
1  4   6   4   1
1  5  10  10   5  1
1  6  15  20  15  6  1
```

A classe `TrianguloDePascal` contém um método `main` que declara, aloca e inicializa o triângulo de Pascal de dez linhas. A classe é mostrada na listagem 11.11.

Listagem 11.11: A classe `TrianguloDePascal`, que demonstra como arrays multidimensionais podem ser irregulares.

```
/**
 * A classe TrianguloDePascal, que demonstra como arrays multidimensionais podem ser
 * irregulares. Esta classe calcula os valores do triângulo de Pascal.
 */
class TrianguloDePascal // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método declara e popula um
  * array triangular (irregular) com os valores do triângulo de Pascal.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Consideramos o número de linhas constante
    int númeroDeLinhas = 10;
    // Declaramos o array bidimensional para conter o triângulo de Pascal, mas só
    // alocamos as linhas do array - as colunas serão alocadas depois, cada uma
    // com um tamanho diferente. Note que a segunda dimensão do array não foi
    // especificada.
    long[][] triânguloDePascal = new long[númeroDeLinhas][];
    // Alocamos cada linha do triângulo separadamente. Cada linha L terá L+2 colunas.
    // Chamamos o operador new para cada linha do triângulo, não para toda a matriz.
    for(int linha=0;linha<númeroDeLinhas;linha++)
      triânguloDePascal[linha] = new long[2+linha];
    // Criamos a primeira linha do triângulo de Pascal usando constantes
    triânguloDePascal[0][0] = 1; triânguloDePascal[0][1] = 1;
    // Executamos um laço para cada linha do triângulo (exceto a primeira)
    for(int linha=1;linha<númeroDeLinhas;linha++)
      {
      // O primeiro elemento de cada linha do triângulo de Pascal é sempre igual a 1
      triânguloDePascal[linha][0] = 1;
      // Cada elemento entre o primeiro e o penúltimo será calculado como a soma do
      // elemento acima e acima à esquerda
      for(int coluna=1;coluna<triânguloDePascal[linha].length-1;coluna++)
        {
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
37          triânguloDePascal[linha][coluna] = triânguloDePascal[linha-1][coluna] +
38                                             triânguloDePascal[linha-1][coluna-1];
39          }
40        // O último elemento de cada linha do triângulo de Pascal é sempre igual a 1
41        triânguloDePascal[linha][triânguloDePascal[linha].length-1] = 1;
42        }
43      // Imprimimos o triângulo de Pascal, usando um laço para as linhas...
44      for(int linha=0;linha<númeroDeLinhas;linha++)
45        {
46        // ... e um para as colunas, que será diferente para cada linha
47        for(int coluna=0;coluna<triânguloDePascal[linha].length;coluna++)
48          System.out.print(triânguloDePascal[linha][coluna]+" ");
49        System.out.println(); // quebra de linhas após cada linha do triângulo
50        }
51      } // fim do método main
52
53    } // fim da classe TrianguloDePascal
```

A declaração, inicialização e manipulação de arrays irregulares é sempre mais complexa e precisa de mais cuidados quando comparada com arrays multidimensionais regulares, mas o uso de memória pode ser menor, o que torna estes arrays interessantes para algumas aplicações.

## 11.4 Processando argumentos da linha de comando

*argumento para o método* `main`

Em todas as aplicações de Java (classes que contém o método `main`) vimos que este método deve ser declarado como recebendo como argumento um array de Strings. Os valores contidos neste array são criados pela máquina virtual Java quando esta tenta executar um programa, usando os argumentos passados na linha de comando do sistema operacional em que esta máquina virtual esteja sendo executado.

Como exemplo, considere uma classe `Programa` que contenha o método `main`. Quando esta classe for executada com o comando `java Programa`, o array de Strings declarado dentro do método `main` terá comprimento zero, ou seja, será vazio, pois nenhum argumento ou parâmetro adicional foi passado para a classe. Se a mesma classe for executada com o comando `java Programa a1 b2 3 4 5`, o array de Strings declarado dentro do método `main` terá comprimento 5, e seus elementos serão `{"a1","b2","3","4","5"}`. É importante notar que mesmo que se alguns elementos do array aparentam ser numéricos, na verdade todos serão instâncias da classe String. Caso seja necessário processar valores numéricos passados pela a linha de comando, estes deverão ser convertidos de Strings para numéricos.

*linha de comando*

A linha de comando é um mecanismo muito prático e eficiente para que usuários passem valores diretamente para o programa sem precisar que este interrompa a execução para pedir valores pelo teclado, especialmente útil para trabalhos em lote ou para passar argumentos que modifiquem o funcionamento do programa.

A classe `CalculadoraDeLinhaDeComando`, mostrada na listagem 11.12, demonstra o uso da linha de comando para criar uma pequena e simples calculadora que recebe três argumentos via linha de comando: o primeiro e o terceiro argumentos devem corresponder a valores inteiros, e o segundo argumento deve ser um caracter correspondente a uma das quatro operações básicas (`'+'`, `'-'`, `'*'` ou `'/'`). O programa usará os argumentos para calcular o valor da expressão, e imprimirá este resultado.

Listagem 11.12: A classe `CalculadoraDeLinhaDeComando`, que implementa uma calculadora simples usando argumentos passados pela linha de comando.

```
/**
 * A classe CalculadoraDeLinhaDeComando, que implementa uma calculadora simples usando
 * argumentos passados pela linha de comando. Esta calculadora somente aceita valores
 * inteiros como argumentos.
 */
class CalculadoraDeLinhaDeComando // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método usa os argumentos
  * passados pela linha de comando, que serão disponibilizados para a aplicação
  * através do array que é o argumento para o método main. Os argumentos serão
  * processados para cálculo da operação entre o primeiro e o terceiro argumentos,
  * onde a operação será definida pelo segundo argumento.
  * @param argumentos os argumentos que foram passados pela linha de comando para
  *         esta aplicação.
  */
  public static void main(String[] argumentos)
    {
    // Primeiro, verificamos se exatamente três argumentos foram passados pela linha
    // de comando, e saímos do programa caso contrário.
    if (argumentos.length != 3)
      {
      System.out.println("Este programa precisa que três argumentos sejam passados "+
                         "pela linha de comando.");
      System.exit(1); // saímos do programa com o código de execução número 1
      }
    // Extraimos um valor inteiro da String correspondente ao primeiro argumento
    int primeiroValor = Integer.parseInt(argumentos[0]);
    // Extraimos o primeiro caracter da String correspondente ao segundo argumento
    char operador = argumentos[1].charAt(0);
    // Extraimos um valor inteiro da String correspondente ao terceiro argumento
    int segundoValor = Integer.parseInt(argumentos[2]);
    // Dependendo do caracter operador, efetuamos a operação
    int resultado = 0; // deve ser inicializada
    switch(operador)
      {
      case '+': resultado = primeiroValor + segundoValor; break;
      case '-': resultado = primeiroValor - segundoValor; break;
      case '*': resultado = primeiroValor * segundoValor; break;
      case '/': resultado = primeiroValor / segundoValor; break;
      }
    // Imprimimos os argumentos passados com espaços entre eles
    for(int índice=0;índice<argumentos.length;índice++)
      System.out.print(argumentos[índice]+" ");
    // Imprimimos o resultado
    System.out.println("= "+resultado);
    } // fim do método main

  } // fim da classe CalculadoraDeLinhaDeComando
```

Os pontos interessantes da classe `CalculadoraDeLinhaDeComando` são:

- Um dos primeiros passos do método `main` é verificar se o número esperado de argumentos foi passado pela linha de comando, e caso o número seja diferente de três, imprimir uma mensagem de erro e sair do programa com o método `exit` da classe `System`. Este método recebe como argumento um valor inteiro que, por convenção, deve ser diferente de zero se o programa terminou inesperadamente ou com erro.
- O array de argumentos é de instâncias da classe String, mas o programa deve processar os valores do tipo `int` correspondentes ao primeiro e terceiro argumentos (de índices 0 e 2). A conversão de Strings para `int`s pode ser feita com o método `parseInt` da classe `Integer` (descrito na seção A.3.5).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

- Similarmente, o operador a ser processado é do tipo `char`, e deve ser extraído da String que é o segundo elemento do array `argumentos`. A extração de caracteres individuais de Strings é feita com o método `charAt` da classe String, que será visto na seção 12.2.
- Ao final da operação, os elementos do array `argumentos` e o resultado são impressos.

*separador de argumentos da linha de comando*

Alguns cuidados importantes devem ser tomados em relação à linha de comando: o array de Strings que é passado como argumento para o método `main` considera que seus elementos são separados por espaços, desta forma, se a classe `CalculadoraDeLinhaDeComando` for chamada com o argumento `"7+5"`, somente uma única String será passada para a aplicação, que terminará com uma mensagem de erro.

*pré-processamento da linha de comando*

Similarmente, quando programas são chamados de terminais, o interpretador de comandos do sistema operacional sendo usado geralmente pré-processa os argumentos passados à procura de caracteres com significados especiais. Desta forma, se uma aplicação qualquer for executada com a linha de comando `*`, o caracter `'*'` será substituído por uma lista contendo todos os nomes de arquivo naquele diretório. Regras similares são usadas para o caracter `'?'` e combinações destes com letras e números. Para passar estes caracteres para a aplicação sem que o pré-processador do interpretador de comando os use, os caracteres devem ser envoltos em aspas duplas. Desta forma, passar os argumentos `7 "*" 5` para a classe `CalculadoraDeLinhaDeComando` dará o resultado esperado, enquando que passar a linha de comando `7 * 5` causará um erro no programa, já que provavelmente o caracter `'*'` será substituído pela lista de nomes de arquivo do diretório corrente, passando assim bem mais que três argumentos para o programa.

## 11.5 Exercícios do capítulo 11

**Exercício 11.1:** ⋆

Após a execução da linha `double[] inversos = new double[100]; inversos[40] = 1./40.;` em um método qualquer, quais das opções abaixo serão verdadeiras ?

**A.** O array `inversos` tem 99 posições.
**B.** `inversos[0]` é igual a `Double.NaN`.
**C.** `inversos[40]` é igual a zero.
**D.** Existem 99 valores no array iguais a zero.
**E.** `inversos[100]` é igual a `null`.

**Exercício 11.2:** ⋆

Após a execução da linha `char[] alfabeto = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'};` em um método qualquer, quais das opções abaixo serão verdadeiras ?

**A.** O array `alfabeto` tem nove posições.
**B.** O quinto elemento do array é o caracter `'F'`.
**C.** O décimo elemento do array é `null`.
**D.** O décimo elemento do array é o caracter espaço.
**E.** O valor de `alfabeto.length` é 8.

**Exercício 11.3:** ⋆

Considerando a declaração `float[] seqüência = new float[25];` quais das declarações abaixo serão corretas (isto é, poderão ser compiladas e executadas sem problemas) ?

**A.** `seqüência[0] = 0;`
**B.** `seqüência[1] = 1;`
**C.** `seqüência[1.5] = 1.5;`
**D.** `seqüência[-1] = -1;`
**E.** `seqüência[23] = "23";`
**F.** `seqüência[24] = 24;`
**G.** `seqüência[25] = 25;`

**Exercício 11.4:** ⋆

Quais das declarações de arrays mostradas abaixo são válidas ?

**A.** `int array = new int[25];`
**B.** `char[] array = new char[0];`
**C.** `float[] array = new double[60];`
**D.** `int[] array = new int[100] ;`
**E.** `double[] array = new double[3.5];`
**F.** `short[] array = new short[-100];`

**Exercício 11.5:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoArray
2    {
3    public static void main(String[] argumentos)
4      {
5      final char[] alfa1 = {'a','b','c','d','e'};
6      char[] alfa2 = {'m','n','o','p','q'};
7      alfa1[2] = '?';
8      alfa2[2] = '?';
9      alfa2 = alfa1;
10     alfa1 = alfa2;
11     }
12   }
```

**Exercício 11.6:** ⋆

Escreva uma aplicação em Java que declare e inicialize um vetor de booleanos (lendo-os do teclado), e calcule quantos elementos são iguais a `true`.

**Exercício 11.7:** ⋆

Escreva uma classe em Java que encapsule um array de 12 bytes, onde cada elemento do array contém o número de dias no mês correspondente, desconsiderando se o ano é bissexto ou não. Por exemplo, o elemento 0 do array (correspondente a Janeiro) deve valer 31. Escreva um método que retorne o valor encapsulado para determinado mês.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 11.8:** ⋆

Qual será o conteúdo dos arrays declarados na aplicação abaixo ao término da execução do método `main` ?

```
1  class ClasseMisteriosa
2    {
3    public static void main(String[] argumentos)
4      {
5      double[] valores = {1,2,3,4,5,6};
6      double[] primeiraCópia = valores;
7      double[] segundaCópia = valores;
8      primeiraCópia[1] = 1;
9      segundaCópia[2] = valores[0]+primeiraCópia[1];
10     primeiraCópia[3] = valores[1]+segundaCópia[2];
11     valores[4] = primeiraCópia[2]+segundaCópia[3];
12     valores[5] = segundaCópia[3]+primeiraCópia[4];
13     }
14   }
```

**Exercício 11.9:** ⋆

Qual será o conteúdo dos arrays declarados na aplicação abaixo ao término da execução do método `main` ?

```
1  class ClasseMisteriosa
2    {
3    public static void main(String[] argumentos)
4      {
5      float[] constantes = {100f,10f,1f,0.1f,0.01f,0.001f};
6      float[] duplicata = constantes;
7      resetaArray(duplicata);
8      }
9    private static void resetaArray(float[] array)
10     {
11     for(int índice=0;índice<array.length;índice++)
12       array[índice] = 0f;
13     }
14   }
```

**Exercício 11.10:** ⋆

O que aconteceria se tentássemos passar um array de valores do tipo `boolean` como primeiro argumento para o método `calculaEMostraSomatória` da classe `CalculoPiQuadradoSobre6` (listagem 11.3) ? E se tentássemos passar um array de valores do tipo `int` ?

**Exercício 11.11:** ⋆

Escreva um programa em Java que crie um array de valores inteiros e que popule o array de forma que o valor do elemento do array seja o dobro de seu índice. Desta forma o quinto elemento do array deverá valer 8 - lembre-se que os índices começam em zero !

**Exercício 11.12:** ⋆

Escreva um programa em Java que crie um array de valores de ponto flutuante de 150 posições, calcule o fatorial de cada uma destas posições e guarde o valor do fatorial no elemento correspondente do array. Ao final do cálculo, a posição 0 do array deve conter 0, a posição 1 deve conter 1, a posição 2 deve conter 2, a posição 3 deve conter 6, a posição 4 deve conter 24 (4!), etc. Use a classe `ProbabilidadeBasica` (listagem 7.7) para os cálculos.

**Exercício 11.13:** ⋆

Escreva um programa que declare um array bidimensional `tabuada` de $10 \times 10$ posições e popule os elementos do array com os valores da tabuada da soma para aquele elemento, de forma que, por exemplo, o elemento `tabuada[7][9]` valha 16.

**Exercício 11.14:** ⋆

Escreva um programa que declare um array bidimensional `tabuada` de $10 \times 10$ posições e popule os elementos do array com os valores da tabuada da multiplicação para aquele elemento, de forma que, por exemplo, o elemento `tabuada[7][9]` valha 63.

**Exercício 11.15:** ⋆

Usando a classe `CalculoPiQuadradoSobre6` (listagem 11.3) como base, escreva uma aplicação em Java que calcule os termos da série que converge para $\frac{\pi^4}{90}$ usando um array. Veja também o exercício 7.32.

**Exercício 11.16:** ⋆

Usando a classe `CalculoPiQuadradoSobre6` (listagem 11.3) como base, escreva uma aplicação em Java que calcule os termos da série que converge para $\frac{\pi^6}{945}$ usando um array. Veja também o exercício 7.33.

**Exercício 11.17:** ⋆

Usando a classe `CalculoPiQuadradoSobre6` (listagem 11.3) como base, escreva uma aplicação em Java que calcule os termos da série que converge para $3/4$ usando um array. Veja também o exercício 7.38.

**Exercício 11.18:** ⋆

Modifique a classe `ArrayDeFloats` (listagem 11.4), criando outro construtor que recebe como argumentos o tamanho do array a ser criado e um valor constante, e inicialize os elementos do array com esta constante.

**Exercício 11.19:** ⋆

O método abaixo pode ser um método da classe `ArrayDeFloats` (listagem 11.4) ? Explique.

```
1 public void mudaTamanho(int novoTamanho)
2   {
3   array.length = novoTamanho;
4   }
```

**Exercício 11.20:** ⋆

Modifique o método `toString` da classe `ArrayDeObjetosGeometricos` (listagem 11.7) para que somente os elementos do array encapsulados que forem diferentes de `null` sejam impressos.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 11.21:** ⋆

Modifique o construtor da classe `MatrizDeDoubles` (listagem 11.9) para que não seja possível a criação de matrizes com uma ou duas das dimensões iguais ou menores do que zero. Se o usuário da classe tentar isto, as dimensões incorretas devem ser consideradas como zero.

**Exercício 11.22:** ⋆

Escreva para a classe `MatrizDeDoubles` métodos que permitam o acesso e a modificação dos valores individuais da matriz encapsulada. *Dica:* veja a classe `ArrayDeFloats` (listagem 11.4).

**Exercício 11.23:** ⋆

Escreva, para a classe `MatrizDeDoubles`, o método `menorValor`, que calcula e retorna o menor elemento da matriz encapsulada.

**Exercício 11.24:** ⋆

Escreva, para a classe `MatrizDeDoubles`, o método `éQuadrada`, que retorna `true` se a matriz for quadrada (isto é, ter o mesmo número de linhas e colunas).

**Exercício 11.25:** ⋆⋆

O exercício 11.12 tem um problema de eficiência: a cada chamada ao método `fatorial` da classe `ProbabilidadeBasica` a série de multiplicações será refeita desde o início, quando o resultado anterior poderia ser aproveitado (por exemplo, o fatorial de 100 é igual a 100 vezes o fatorial de 99). Modifique o programa de forma que ao invés de chamar o método `fatorial` a cada passo, o laço que preenche o array simplesmente use o valor do elemento anterior e multiplique-o pelo índice, dando assim o valor do elemento sendo calculado.

**Exercício 11.26:** ⋆⋆

Usando a classe `CalculoPiQuadradoSobre6` (listagem 11.3) como base, escreva uma aplicação em Java que calcule os termos da série que converge para $1/2$ usando um array. Veja também o exercício 7.37.

**Exercício 11.27:** ⋆⋆

Usando a classe `CalculoPiQuadradoSobre6` (listagem 11.3) como base, escreva uma aplicação em Java que calcule os termos da série que converge para o logaritmo de 2 na base natural usando um array. Veja também o exercício 7.40.

**Exercício 11.28:** ⋆⋆

Usando a classe `CalculoPiQuadradoSobre6` (listagem 11.3) como base, escreva uma aplicação em Java que calcule os termos da série que converge para $\frac{\pi^3}{32}$ usando um array. Veja também o exercício 7.45.

**Exercício 11.29:** ⋆⋆

Escreva um programa que declare um array de cinco dimensões `tabuada` de $10 \times 10 \times 10 \times 10 \times 10$ posições e popule os elementos do array com os valores da tabuada da multiplicação para aquele elemento, de forma que, por exemplo, o elemento `tabuada[6][5][2][1][4]` valha 240. *Dica:* use o exercício 11.14 como base.

**Exercício 11.30:** ⋆⋆

Considerando o exercício 11.12, escreva uma classe `Fatorial` que contenha um construtor e um método `valor()` que recebe um valor inteiro como argumento e que retorna um valor de ponto flutuante. O construtor, quando chamado, deverá alocar um array de 150 posições e calcular o fatorial de 0 a 149, guardando os resultados no array. O método `valor()`, quando chamado, deverá retornar o valor do fatorial correspondente ao valor passado como argumento.

**Exercício 11.31:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) um método `éIgual` que receba outra instância da classe como argumento e retorne `true` se os tamanhos e valores do array encapsulado e do passado como argumento forem iguais.

**Exercício 11.32:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) um método `soma` que some (acumule) aos elementos do array encapsulado uma constante do tipo `float` que será passada como argumento.

**Exercício 11.33:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) um método `multiplica` que multiplique todos os elementos do array encapsulado por uma constante do tipo `float` que será passada como argumento.

**Exercício 11.34:** ⋆⋆

Escreva um método `imprimeChefes` para a classe `Equipe` (listagem 11.6) que receba o array de instâncias da classe `Funcionario` como argumento e imprima somente os elementos que forem instâncias da classe `ChefeDeDepartamento`. O método não deverá imprimir instâncias que sejam iguais a `null`.

**Exercício 11.35:** ⋆⋆

Usando o exercício 11.34 como base, escreva um método `imprimeEquipe` que imprima as instâncias não-nulas do array de instâncias da classe `Funcionario`, que deve ser passado como argumento. A impressão deve ser feita de forma que os chefes de departamento sejam impressos antes dos funcionários.

**Exercício 11.36:** ⋆⋆

Crie na classe `MatrizDeDoubles` (listagem 11.9) o método `toString` que retorne uma string contendo os elementos da matriz. Faça com que esta string tenha várias linhas, correspondente às linhas da matriz. *Dica:* o caracter `'\n'` significa quebra de linha, e pode estar presente no meio de strings.

**Exercício 11.37:** ⋆⋆

Crie na classe `MatrizDeDoubles` (listagem 11.9) um método `éIgual` que receba outra instância da classe como argumento e retorne `true` se os tamanhos e valores da matriz encapsulada e da passada como argumento forem iguais.

**Exercício 11.38:** ⋆⋆

Crie na classe `MatrizDeDoubles` (listagem 11.9) um método `soma` que some (acumule) aos elementos da matriz encapsulada uma constante do tipo `double` que será passada como argumento.

**Exercício 11.39:** ⋆⋆

Crie na classe `MatrizDeDoubles` (listagem 11.9) um método `multiplica` que multiplique todos os elementos da matriz encapsulada por uma constante do tipo `double` que será passada como argumento.

**Exercício 11.40:** ⋆⋆

Escreva uma classe que encapsule uma matriz de tamanho $2 \times 2$ de valores do tipo float ou double, usando um array de duas dimensões. Nesta classe, escreva um método que calcule o determinante da matriz encapsulada, e um método que permita a impressão formatada dos seus valores (duas linhas com dois valores cada).
*Dica:* Se a matriz M é dada por

$$\begin{pmatrix} x_{00} & x_{01} \\ x_{10} & x_{11} \end{pmatrix}$$

então o determinante é calculado como $(x_{00} \times x_{11}) - (x_{01} \times x_{10})$.

**Exercício 11.41:** ⋆⋆

Usando o exercício 11.40 como base, escreva uma classe que encapsule uma matriz $3 \times 3$ usando um array de duas dimensões. Nesta classe, escreva um método que calcule o determinante da matriz encapsulada, e um método que permita a impressão formatada dos seus valores (duas linhas com dois valores cada).
*Dica:* Se a matriz M é dada por

$$\begin{pmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{pmatrix}$$

então o determinante é calculado como
$(x_{00} \times x_{11} \times x_{12}) + (x_{01} \times x_{12} \times x_{20}) + (x_{02} \times x_{10} \times x_{21}) - (x_{00} \times x_{12} \times x_{21}) - (x_{01} \times x_{10} \times x_{22}) - (x_{02} \times x_{11} \times x_{20})$

**Exercício 11.42:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) um método `quantosPositivos` que retorne um valor do tipo `int` correspondente ao número de elementos positivos (acima de zero) que existam no array, e também um método `quantosNegativos`, que retorne quantos elementos abaixo de zero existem no array.

**Exercício 11.43:** ⋆⋆

Baseado no exercício 11.42, crie na classe `ArrayDeFloats` os métodos `quantosZeros` e `quantosInfinitos` que retornem o número de elementos do array encapsulado que sejam respectivamente iguais a zero e infinitos (positivos ou negativos). *Dica:* use as constantes `Float.POSITIVE_INFINITY` e `Float.NEGATIVE_INFINITY` para verificar se o valor é infinito.

**Exercício 11.44:** ⋆⋆

Crie na classe `ArrayDeFloats` o método `existe`, que recebe um valor do tipo `float` como argumento e retorna o booleano `true` se o valor passado como argumento existir no array encapsulado.

**Exercício 11.45:** ⋆⋆

Crie na classe `ArrayDeFloats` o método `média`, que retorna a média dos valores encapsulados no array. *Dica:* valores infinitos, se houverem, não devem ser considerados.

**Exercício 11.46:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) um outro método `soma` que receba uma outra instância da classe `ArrayDeFloats` como argumento e acumule os valores do array passado como argumento ao array encapsulado. Esta operação somente poderá ser feita se os arrays tiverem o mesmo tamanho, se o tamanho for diferente o array encapsulado não deverá ser modificado. Veja também o exercício 11.32.

**Exercício 11.47:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) um outro método `multiplica` que receba uma outra instância da classe `ArrayDeFloats` como argumento e multiplique cada valor do array passado como argumento ao valor correspondente do array encapsulado. Esta operação somente poderá ser feita se os arrays tiverem o mesmo tamanho, se o tamanho for diferente o array encapsulado não deverá ser modificado. Veja também os exercícios 11.33 e 11.46.

**Exercício 11.48:** ⋆⋆

Considere a classe `ArrayDeFloats` (listagem 11.4). Crie nesta classe o método `troca` que recebe dois valores inteiros como argumentos e troca os valores nas posições especificadas pelos argumentos. Por exemplo, se os valores do array encapsulado forem $\{3.6, 2.7, 8.0, 9.2, 1.5, 4.1\}$ e o método `troca` for chamado com os argumentos 1 e 4, os elementos nestes índices serão trocados de forma que os valores do array encapsulado serão $\{3.6, 1.5, 8.0, 9.2, 2.7, 4.1\}$ (note os valores que foram trocados). *Dica:* só é possível fazer esta troca usando um valor temporário do tipo `float`.

**Exercício 11.49:** ⋆⋆

Usando o exercício 11.48 como base, reescreva o método `troca` para que este somente troque os valores se o valor na posição do primeiro índice for **menor** do que o valor na posição do segundo índice. Por exemplo, se os valores no array encapsulado forem $\{3.6, 2.7, 8.0, 9.2, 1.5, 4.1\}$, uma chamada ao método `troca(0,1)` trocaria os valores nas posições 0 e 1, mas uma chamada ao método `troca(1,2)` não trocaria os valores de índices 1 e 2 de posição.

**Exercício 11.50:** ⋆⋆

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `éDiagonal` que retorne `true` se a matriz for diagonal (isto é, se a matriz for quadrada, e se todos os elementos da diagonal forem iguais a um, e todos os outros elementos forem iguais a zero). *Dica:* use o método `éQuadrada`, criado como resposta ao exercício 11.24.

**Exercício 11.51:** ⋆⋆

Escreva, para a classe `MatrizDeDoubles`, o método `elementosSãoIguais` que retorne `true` se todos os elementos da matriz tiverem o mesmo valor. *Dica:* é possível escrever este método de maneira simplificada usando os métodos `maiorValor` e `menorValor` desta classe.

**Exercício 11.52:** ⋆⋆

Escreva, para a classe `ArrayDeFloats`, o método `clona`, que deve retornar uma outra instância da classe `ArrayDeFloats` com os mesmos valores da instância que executar o método.

**Exercício 11.53:** ⋆⋆

Escreva, para a classe `ArrayDeFloats`, o método `clonaParaArray`, que deve retornar um array de valores do tipo `float` com os mesmos valores da instância que executar o método.

**Exercício 11.54:** ⋆⋆

Escreva, para a classe `ArrayDeFloats`, o método `clonaRegião`, que deve retornar uma outra instância da classe `ArrayDeFloats` correspondente a uma região do array original, cujos delimitadores devem ser passados como argumentos. Por exemplo, se o array encapsulado é $\{9,3,6,1,0,3,7,2\}$ e o método for chamado como `clonaRegião(2,4)` uma nova instância da classe `ArrayDeFloats` contendo quatro elementos a partir do elemento de índice dois deve ser criada (neste exemplo, $\{6,1,0,3\}$).

**Exercício 11.55:** ⋆⋆

Escreva, para a classe `ArrayDeFloats`, o método `clonaRegiãoParaArray`, que deve retornar um array de valores do tipo `float` com os valores de uma região do array original, cujos delimitadores devem ser passados como argumentos. Veja também o exercícios 11.53 e 11.54.

**Exercício 11.56:** ⋆⋆

Escreva, para a classe `MatrizDeDoubles`, o método `clona`, que deve retornar uma outra instância da classe `MatrizDeDoubles` com os mesmos valores da instância que executar o método.

**Exercício 11.57:** ⋆⋆

Escreva, para a classe `MatrizDeDoubles`, o método `clonaParaMatriz`, que deve retornar uma matriz de valores do tipo `double` com os mesmos valores da instância que executar o método.

**Exercício 11.58:** ⋆⋆

Escreva uma classe `MatrizDeCaracteres` que encapsule uma matriz de caracteres (valores do tipo `char`). As dimensões desta matriz devem ser especificadas através do construtor da classe. O construtor da classe deve também inicializar a matriz com caracteres aleatórios (que podem ser obtidos usando-se a expressão `(char)('A'+Math.random()*26)`). Escreva também um método `toString` para esta classe.

**Exercício 11.59:** ⋆⋆

Melhore a calculadora de linha de comando implementada pela classe `CalculadoraDeLinhaDeComando` (listagem 11.12) para que os argumentos passados sejam de ponto flutuante e que erros de divisão por zero sejam indicados. Faça também com que, caso um operador diferente de `'+'`, `'-'`, `'*'` e `'/'` seja passado, uma mensagem de erro seja impressa. *Dica:* Strings podem ser convertidas para valores do tipo `double` com o método `parseDouble` da classe `Double`.

**Exercício 11.60:** ⋆⋆

Escreva uma classe `Menu` em Java que encapsule um array de strings. Nesta classe, crie o método `escolhe` que mostra as strings com um número na frente, espere o usuário digitar um número e retorne a string correspondente àquele número. *Dicas:* faça com que o usuário seja impossibilitado de digitar números fora da faixa com um laço `while`, e faça com que os números a serem impressos antes das strings estejam na faixa $1-N$, onde N é o tamanho do array. Desta forma a escolha fica mais amigável para o usuário.

**Exercício 11.61:** ⋆⋆

Escreva para a classe `ArrayDeObjetosGeometricos` um método `clona`, que deve retornar uma outra instância da classe `ArrayDeObjetosGeometricos` com os mesmos valores da instância que executar o método.

**Exercício 11.62:** ⋆⋆⋆

Escreva na classe `ArrayDeFloats` o método `maisPróximo`, que recebe um valor do tipo `float` como argumento e retorna o valor do array encapsulado que seja mais próximo (ou seja, cuja diferença seja a menor) do valor passado. Por exemplo, se o array encapsulado for $\{0,-2,-4,10\}$ e o argumento para o método for 6, o método deverá retornar 10. *Dica:* valores infinitos, positivos ou negativos, não devem ser considerados.

**Exercício 11.63:** ⋆⋆⋆

Escreva na classe `ArrayDeFloats` o método `reverte`, que reverte a ordem dos elementos do array encapsulado, de forma que o primeiro passe a ser o último e vice-versa. Por exemplo, se o array encapsulado for $\{9,9,2,7,0,5\}$, depois da execução do método ele será $\{5,0,7,2,9,9\}$. *Dica:* existem duas abordagens para a solução deste problema, uma que modifica os valores do array encapsulado e outra que cria uma nova instância, considere implementar as duas.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 11.64:** ⋆⋆⋆

Escreva na classe `ArrayDeFloats` o método `éCrescente`, que verifica se os elementos de um array estão ordenados crescentemente, comparando cada elemento do array com seu próximo, retornando `true` se todos os elementos forem menores que os seus respectivos próximos ou `false` se qualquer um for maior do que o próximo.

**Exercício 11.65:** ⋆⋆⋆

Escreva na classe `ArrayDeFloats` o método `éDecrescente`, que verifica se os elementos de um array estão ordenados decrescentemente, comparando cada elemento do array com seu próximo, retornando `true` se todos os elementos forem maiores que os seus respectivos próximos ou `false` se qualquer um for menor do que o próximo. Note que este método não é o complemento lógico do método `éCrescente` (exercício 11.64).

**Exercício 11.66:** ⋆⋆⋆

Escreva na classe `ArrayDeFloats` os métodos `compõeMínimos` e `compõeMáximos`, que fazem respectivamente a composição dos valores mínimos e máximos entre o array encapsulado e um outro array, passado como argumento, retornando uma nova instância da classe com o array composto. A composição é feita comparando cada elemento do array encapsulado com o elemento correspondente do array passado como argumento, e considerando o mínimo ou máximo, dependendo do método. Por exemplo, se o array encapsulado for $\{1,3,5,6,7,9,11\}$ e o passado como argumento for $\{2,8,5,7,3,1,2\}$ o array composto dos mínimos deverá ser $\{1,3,5,6,3,1,2\}$ e o composto dos máximos deverá ser $\{2,8,5,7,7,9,11\}$. Os métodos devem retornar `null` se os tamanhos dos arrays forem diferentes.

**Exercício 11.67:** ⋆⋆⋆

Usando o exercício 11.66 como base, escreva os métodos `compõeMínimos` e `compõeMáximos` para a classe `MatrizDeDoubles`.

**Exercício 11.68:** ⋆⋆⋆

Escreva na classe `ArrayDeFloats` um método `concatena` que recebe uma outra instância da classe `ArrayDeFloats` como argumento, e retorna uma nova instância da mesma classe contendo os valores concatenados, isto é, os dois arrays como um só. Por exemplo, se o array encapsulado for $\{3,5,7,9,11\}$ e o array passado como argumento for $\{1,1,7,13\}$ o método deverá retornar o array $\{3,5,7,9,11,1,1,7,13\}$.

**Exercício 11.69:** ⋆⋆⋆

Escreva versões sobrecarregadas do método `concatena` para a classe `ArrayDeFloats` (veja o exercício 11.68), que recebam dois, três e quatro instâncias da classe `ArrayDeFloats` como argumentos. *Dica:* uma solução simples usa os próprios métodos `concatena` para efetuar a concatenação.

**Exercício 11.70:** ★★★

Crie na classe `ArrayDeFloats` (listagem 11.4) um método `distânciaEuclideana` que retorne um valor do tipo float que seja igual à distância Euclideana acumulada entre o array encapsulado e de uma outra instância de `ArrayDeFloats` passada como argumento. A distância Euclideana entre dois arrays numéricos é definida como sendo a raiz quadrada das somas dos quadrados das diferenças de seus elementos. Por exemplo, se o array encapsulado for $\{1,3,0\}$ e o passado como argumento for $\{1,9,-4\}$ a distância Euclideana será igual a $\sqrt{(1-1)^2+(3-9)^2+(0-(-4))^2} = \sqrt{0+36+16} \approx 7.2111$. O método deverá retornar a constante `Float.NaN` caso os arrays sejam de tamanhos diferentes. *Dica:* a raiz quadrada de uma expressão qualquer pode ser calculada com o método `sqrt` da classe `Math`, descrito na seção 13.3.3.

**Exercício 11.71:** ★★★

Crie na classe `ArrayDeFloats` (listagem 11.4) um método `produtoEscalar` que retorne um valor do tipo float que seja o produto escalar do array encapsulado e de uma outra instância de `ArrayDeFloats` passada como argumento. Por exemplo, se o array encapsulado for $\{9,2,-6,7,0\}$ e o passado como argumento for $\{1,-4,5,9,2\}$ o produto escalar será $9\times1-2\times4-6\times5+7\times9+0\times2=34$. O método deverá retornar a constante `Float.NaN` caso os arrays sejam de tamanhos diferentes.

**Exercício 11.72:** ★★★

Crie na classe `ArrayDeFloats` o método `existeQualquer`, que recebe uma outra instância da classe `ArrayDeFloats` como argumento e retorna o booleano `true` se qualquer um dos elementos do array passado como argumento existir no array encapsulado.

**Exercício 11.73:** ★★★

Crie na classe `ArrayDeFloats` o método `existemTodos`, que recebe uma outra instância da classe `ArrayDeFloats` como argumento e retorna o booleano `true` se todos os elementos do array passado como argumento existirem no array encapsulado, em qualquer ordem e independentemente de repetições.

**Exercício 11.74:** ★★★

Crie na classe `MatrizDeDoubles` (listagem 11.9) um outro método `soma` que receba uma outra instância da classe `MatrizDeDoubles` como argumento e acumule os valores da matriz passada como argumento à matriz encapsulada. Esta operação somente poderá ser feita se as matrizes tiverem o mesmo tamanho, se o tamanho for diferente a matriz encapsulada não deverá ser modificada. Veja também os exercícios 11.46 e 11.38.

**Exercício 11.75:** ★★★

Crie na classe `MatrizDeDoubles` (listagem 11.9) um outro método `multiplica` que receba uma outra instância da classe `MatrizDeDoubles` como argumento e multiplique cada valor da matriz passada como argumento ao valor correspondente da matriz encapsulada. Esta operação somente poderá ser feita se as matrizes tiverem o mesmo tamanho, se o tamanho for diferente a matriz encapsulada não deverá ser modificada. Veja também os exercícios 11.47 e 11.39.

**Exercício 11.76:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles`, o método `clonaRegião`, que deve retornar uma outra instância da classe `MatrizDeDoubles` correspondente a uma região da matriz original, cujos delimitadores devem ser passados como argumentos. Por exemplo, se a matriz encapsulada for

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix}$$

e o método for chamado como `clonaRegião(0,1,2,3)` uma nova matriz $2 \times 3$ deve ser criada, e preenchida com os valores a partir da linha 0 e coluna 1 da matriz encapsulada, ou seja, a matriz

$$\begin{pmatrix} 2 & 3 & 4 \\ 7 & 8 & 9 \end{pmatrix}$$

deverá ser criada e retornada.

**Exercício 11.77:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles`, o método `clonaRegiãoParaMatriz`, que deve retornar uma matriz de valores do tipo `double` contendo uma região da matriz original, cujos delimitadores devem ser passados como argumentos. Veja também os exercícios 11.55 e 11.76

**Exercício 11.78:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `éTriangularSuperior` que retorne `true` se a matriz for triangular superior, isto é, se os elementos nas posições $(l,c)$ forem iguais a zero para $l > c$, onde $l$ é o índice da linha da matriz e $c$ o índice da coluna da matriz.

**Exercício 11.79:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `éTriangularInferior` que retorne `true` se a matriz for triangular inferior, isto é, se os elementos nas posições $(l,c)$ forem iguais a zero para $c > l$, onde $l$ é o índice da linha da matriz e $c$ o índice da coluna da matriz.

**Exercício 11.80:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `elementosSãoDiferentes` que retorne `true` se todos os elementos da matriz forem diferentes, isto é, se não existirem elementos duplicados na matriz. Note que este método não é o inverso lógico do método `elementosSãoIguais`, pedido no exercício 11.51.

**Exercício 11.81:** ⋆⋆⋆

Escreva para a classe `ArrayDeObjetosGeometricos` um método `clonaInicializados`, que deve retornar uma outra instância da classe `ArrayDeObjetosGeometricos` contendo somente os objetos não-nulos do array encapsulado.

**Exercício 11.82:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `trocaLinhas` que troque os valores de duas linhas da matriz. Os índices das duas linhas devem ser passados como argumentos ao método. Por exemplo, se a matriz encapsulada é

$$\begin{pmatrix} 2 & -3 & 0 \\ 1 & -1 & 11 \\ -8 & 2 & 5 \\ 17 & 3 & -9 \end{pmatrix}$$

e o método é chamado como `trocaLinhas(2,3)` a matriz encapsulada, após o processamento, será

$$\begin{pmatrix} 2 & -3 & 0 \\ 1 & -1 & 11 \\ 17 & 3 & -9 \\ -8 & 2 & 5 \end{pmatrix}.$$

**Exercício 11.83:** ⋆⋆⋆

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `trocaColunas` que troque os valores de duas colunas da matriz. Os índices das duas colunas devem ser passados como argumentos ao método. Por exemplo, se a matriz encapsulada é

$$\begin{pmatrix} 2 & -3 & 0 \\ 1 & -1 & 11 \\ -8 & 2 & 5 \\ 17 & 3 & -9 \end{pmatrix}$$

e o método é chamado como `trocaColunas(1,3)` a matriz encapsulada, após o processamento, será

$$\begin{pmatrix} 0 & -3 & 2 \\ 11 & -1 & 1 \\ 5 & 2 & -8 \\ -9 & 3 & 17 \end{pmatrix}.$$

**Exercício 11.84:** ⋆⋆⋆

Crie uma classe que implementa um *array de escrita única* de valores de um tipo numérico qualquer. Um array de escrita única é um array cujos elementos só possam ser modificados uma única vez. Para a implementação, devemos ter, para cada elemento do array, um valor booleano que diz se o elemento pode ser modificado ou não. Quando instâncias desta classe forem criadas, todos os elementos do array poderão ser modificados, mas assim que um elemento for modificado pela primeira vez, o seu valor booleano associado será modificado de forma que da próxima vez que o elemento for modificado nada ocorrerá.

**Exercício 11.85:** ⋆⋆⋆

Escreva a classe que implementa um *array de escrita única* (exercício 11.84) como sendo uma classe herdeira da classe `ArrayDeFloats`. Quais métodos devem ser sobrepostos ?

**Exercício 11.86:** ⋆⋆⋆

Usando o exercício 11.84 como base, crie uma classe que implementa um array cujos elementos só podem ser modificado um certo número de vezes. Quando instâncias desta classe forem criadas, elas devem receber um valor que diz quantas vezes um dado elemento do array encapsulado pode ser modificado. *Dica:* o valor deve ser o mesmo para todos os elementos do array, mas cada elemento individual do array deverá poder ser modificado o número de vezes que for especificado.

**Exercício 11.87:** ⋆⋆⋆

Escreva para a classe `ArrayDeObjetosGeometricos` (listagem 11.7) os métodos `calculaMaiorÁrea` e `calculaMenorÁrea` que calculem e retornem, respectivamente, a maior e menor área dos objetos contidos no array encapsulado. *Dica:* verifique se o elemento é `null` antes de tentar calcular a sua área. Veja também a classe `ArrayDeFloats` (listagem 11.4) como referência.

**Exercício 11.88:** ⋆⋆⋆

Escreva para a classe `ArrayDeObjetosGeometricos` (listagem 11.7) os métodos `calculaMaiorPerímetro` e `calculaMenorPerímetro` que calculem e retornem, respectivamente, o maior e menor perímetro dos objetos contidos no array encapsulado. *Dica:* verifique se o elemento é `null` antes de tentar calcular o seu perímetro. Veja também a classe `ArrayDeFloats` (listagem 11.4) como referência.

**Exercício 11.89:** ⋆⋆⋆

Escreva para a classe `ArrayDeObjetosGeometricos` (listagem 11.7) os métodos `somaDasÁreas` e `somaDosPerímetros` que calculem e retornem, respectivamente, a soma das áreas e dos perímetros dos objetos contidos no array encapsulado. *Dica:* verifique se o elemento é `null` antes de tentar calcular a sua área ou perímetro. Veja também a classe `ArrayDeFloats` (listagem 11.4) como referência.

**Exercício 11.90:** ⋆⋆⋆

Escreva para a classe `ArrayDeObjetosGeometricos` (listagem 11.7) os métodos `arrayDeÁreas` e `arrayDePerímetros` que retornem, cada um, uma instância da classe `ArrayDeFloats` com o mesmo número de elementos, e onde cada elemento do array de valores de ponto flutuante contenha a área (para o método `arrayDeÁreas`) e perímetro (para o método `arrayDePerímetros`) do objeto geométrico correspondente.

**Exercício 11.91:** ⋆⋆⋆

Escreva uma classe `Banco` que encapsule um array de instâncias da classe `ContaBancariaSimplificada` (exercício 2.53). Escreva, para esta classe, um método `total` que calcule e retorne o total dos saldos de todas as contas bancárias encapsuladas como elementos do array.

**Exercício 11.92:** ★★★

Crie uma classe `EntradaEmAgenda` que contenha os dados necessários para armazenar uma entrada de agenda (hora, dia, mês, ano, assunto). Crie nesta classe, além do construtor e do método `toString`, um método `éNoDia` que recebe valores de dia, mês e ano e retorna `true` se o dia, mês e ano daquela instância da classe forem iguais aos argumentos passados. Crie um método similar chamado `éNoMês` que recebe valores de mês e ano somente e retorna `true` se o mês e ano daquela instância da classe forem iguais aos argumentos passados.
Crie também uma classe `Agenda` que encapsule uma agenda de compromissos, que será representado por um array de instâncias da classe `EntradaEmAgenda`. Crie um método `listaDia` que recebe valores de dia, mês e ano e lista **todas** as instâncias de `EntradaEmAgenda` que caem naquele dia, mês e ano, e um método `listaMês` que recebe valores de mês e ano e lista **todas** as instâncias de `EntradaEmAgenda` que caem naquele mês e ano.

**Exercício 11.93:** ★★★

Usando a classe `CalculadoraDeLinhaDeComando` (listagem 11.12) como base, escreva uma aplicação em Java que processe a linha de comando, recebendo três ou mais argumentos, de forma que o primeiro deva ser o operador `'+'` ou `'*'`, e os argumentos do segundo em diante devam ser valores numéricos. A aplicação deve efetuar a soma ou multiplicação de todos os argumentos passados e mostrar o resultado. Se, por exemplo, os argumentos `+ 2 4 1 5` forem passados, a aplicação deverá imprimir o resultado `12`. Se os argumentos `* 2 4 1 5` forem passados, a aplicação deverá imprimir o resultado `40`.

**Exercício 11.94:** ★★★

Modifique o resultado do exercício 11.59 para que várias operações possam ser feitas simultaneamente, de forma que ao passamos a linha de comando `2 + 3 4 "*" 8 9 - 3 10 / 4` quatro operações sejam feitas, e a aplicação mostre o resultados `5 32 6 2.5`. *Dica:* o número de argumentos deve ser múltiplo de três, o que pode ser verificado comparando-se o resultado a operação resto com zero zero (`10 % 3 = 1; 9 % 3 = 0`).

**Exercício 11.95:** ★★★

Usando o exercício 11.60 como base, modifique a classe `Menu` para que ela encapsule, ao invés de um array de strings, um array de instâncias da classe `ItemDeMenu`. Esta classe deve encapsular uma string contendo o nome do item de restaurante e um valor de ponto flutuante contendo o preço daquele item. Modifique o método `escolhe` adequadamente. É viável fazer isto usando o mecanismo de herança ?

**Exercício 11.96:** ★★★

Modifique o resultado do exercício 11.95, de forma que a classe `Menu` encapsule também um array de valores inteiros, com o mesmo número de elementos do array de instâncias da classe `ItemDeMenu`, correspondente ao número de pedidos de cada item. Este array adicional servirá para acumular pedidos, ou seja, para cada item pedido no método `escolhe`, o elemento correspondente neste array será incrementado em um. Escreva na classe `Menu` um método que imprima um recibo do que foi pedido, incluindo os totais parciais (valor de cada item vezes o número de itens pedidos).

**Exercício 11.97:** ⋆⋆⋆

Escreva uma classe `ArrayDeFloatsUnicos` que herde da classe `ArrayDeFloats` e implemente um array onde valores repetidos não podem existir. *Dica:* cada vez que tentarmos inserir um valor neste array, o método `modifica` deve verificar se o valor existe, só modificando ou inserindo se este ainda não existir no array encapsulado.

**Exercício 11.98:** ⋆⋆⋆

Escreva uma classe `ArrayDePontos2D`, no molde da classe `ArrayDeFloats` (listagem 11.4) que encapsule um array de instâncias da classe `Ponto2D` (listagem 3.2). Esta classe deve ter os seguintes métodos:

- Construtor, que recebe como argumento um número máximo de instâncias da classe `Ponto2D` que serão encapsuladas pela classe,
- `tamanho`, que retorna o tamanho do array encapsulado,
- `modifica`, que recebe como argumentos um valor inteiro (posição) e uma instância da classe `Ponto2D`, e faz com que a instância naquela posição do array passe a ser a passada como argumento,
- `valor`, que recebe como argumento um valor inteiro (posição) e retorna a instância armazenada naquela posição do array,
- `toString`, que retorna uma única string contendo todas os valores das instâncias no array encapsulado na classe.

Para facilitar a criação da classe, considere que a classe `Ponto2D` tem ao menos um construtor que recebe argumentos (veja o exercício 4.24) e métodos para recuperar as coordenadas $x$ e $y$ do ponto.

**Exercício 11.99:** ⋆⋆⋆

Escreva, para a classe `ArrayDePontos2D` (exercício 11.98), um método `pontoSuperior` que retorne o ponto cuja coordenada `y` é a maior dentre as encapsuladas no array; um método `pontoInferior` que retorne o ponto cuja coordenada `y` é a menor dentre as encapsuladas no array; um método `pontoMaisÀEsquerda` que retorne o ponto cuja coordenada `y` é a menor dentre as encapsuladas no array; e um método `pontoMaisÀDireita` que retorne o ponto cuja coordenada `y` é a maior dentre as encapsuladas no array.

**Exercício 11.100:** ⋆⋆⋆

Escreva, para a classe `ArrayDePontos2D`, um método `retânguloEnvolvente` que retorne uma instância da classe `Retangulo` (exercícios 2.50 e 4.27) contendo o retângulo envolvente de todos os pontos do array encapsulado, de forma que todos os pontos do array estejam localizados dentro do retângulo (veja a figura 11.2, onde o retângulo é definido pelas coordenadas $(x1,y1)$ e $(x2,y2)$). *Dica:* use os métodos criados no exercício 11.99.

**Exercício 11.101:** ⋆⋆⋆

Escreva, para a classe `ArrayDePontos2D` (listagem 11.98), um método `centro` que calcule e retorne o ponto central de todos os pontos diferentes que estejam encapsulados no array, isto é, o ponto localizado nas médias das coordenadas $x$ e $y$.

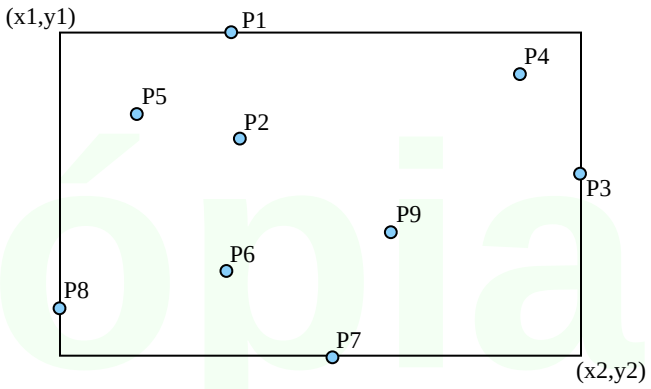



Figura 11.2: Retângulo envolvente de uma série de pontos

**Exercício 11.102:** ⋆⋆⋆

Escreva, para a classe `ArrayDePontos2D`, um método `dispersão` que calcule e retorne a dispersão de todos os pontos do array. A dispersão é calculada como a soma de todas as distâncias entre o ponto central e cada ponto do array. Use o resultado do exercício 11.101.

**Exercício 11.103:** ⋆⋆⋆

Escreva, para a classe `ArrayDeFloats` (listagem 11.4) um método `aplicaMédiaMóvel` que transforme cada elemento do array na média dos valores dos três vizinhos mais próximos, retornando uma nova instância da classe contendo o array modificado. Se houverem valores infinitos ou `NaN` no array, o método deve retornar `null`. Os três vizinhos mais próximos de um elemento são o próprio elemento, o elemento que estiver antes deste e o que estiver depois deste. Caso não existam elementos antes ou depois do elemento sendo considerado (por exemplo, no início ou final do array), somente os que existirem devem ser considerados, desta forma o primeiro elemento do array deverá ser substituido pela média dos valores dele mesmo e do segundo elemento do array. Para ilustrar o algoritmo, uma simulação é mostrada abaixo, que usa como exemplo o array $\{62.0, 64.8, 68.8, 64.7, 61.2, 57.1, 56.3, 60.0, 61.0\}$:

| Elemento original | Média calculada como | Elemento resultante |
|---:|:---:|---:|
| 62.0 | (62.0+64.8) / 2 | 63.4 |
| 64.8 | (62.0+64.8+68.8) / 3 | 65.2 |
| 68.8 | (64.8+68.8+64.7) / 3 | 66.1 |
| 64.7 | (68.8+64.7+61.2) / 3 | 64.9 |
| 61.2 | (64.7+61.2+57.1) / 3 | 61.0 |
| 57.1 | (61.2+57.1+56.3) / 3 | 58.2 |
| 56.3 | (57.1+56.3+60.0) / 3 | 57.8 |
| 60.0 | (56.3+60.0+61.0) / 3 | 59.1 |
| 61.0 | (60.0+61.0) / 2 | 60.5 |

*Dica:* para resolver este problema, é necessária a criação de um array temporário do tamanho do array original - se as operações de média forem feitas diretamente no array encapsulado, os valores deste serão corrompidos para o cálculo dos próximos valores.

**Exercício 11.104:** ★★★★

Crie uma classe que represente um jogo da velha, usando uma matriz de duas dimensões para representar as posições do jogo. A matriz deve ser alocada no construtor da classe, ter o tamanho $3 \times 3$ e ser de um tipo que suporte três estados possíveis: vazio, preenchido com `'O'` e preenchido com `'X'`. A classe deve poder ser usada para jogos com dois jogadores.
*Dica:* A classe deve ter os seguintes métodos:

- `jogaO`, que aceita dois valores que são as coordenadas onde um `'O'` será jogado, e marca na matriz a posição **somente** se esta estiver livre.
- `jogaX`, que aceita dois valores que são as coordenadas onde um `'X'` será jogado, e marca na matriz a posição **somente** se esta estiver livre.
- `verifica`, que verifica a matriz para ver se existe algum ganhador (este método deve verificar se existem três marcas iguais que não sejam vazias em uma horizontal, vertical ou diagonal da matriz).
- `toString`, que retornará uma string com a representação gráfica do jogo com as posições atuais.

Escreva também um programa que use a classe. Este programa deve executar um laço no qual fica perguntando as posições para os jogadores alternadamente, enquanto não houver vitória, desistência ou acabarem as posições vazias da matriz.

**Exercício 11.105:** ★★★★

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), um método `multiplica`, que aceite outra instância da classe `MatrizDeDoubles` como argumento e multiplique a matriz encapsulada pela passada como argumento, retornando ma nova matriz.
A multiplicação de uma matriz $A$ de dimensões $A_l$x$A_c$ por uma matriz $B$ de dimensões $B_l$x$B_c$ só pode ser efetuada se os valores $A_c$ e $B_l$ forem iguais, ou seja, se o número de colunas da matriz $A$ for igual ao número de linhas da matriz $B$. O resultado será uma matriz $C$ de dimensões $A_l$x$B_c$. O esquema gráfico da multiplicação é mostrado abaixo:

$$\begin{bmatrix} A_{(0,0)} & A_{(0,1)} & A_{(0,2)} & A_{(0,3)} \\ A_{(1,0)} & A_{(1,1)} & A_{(1,2)} & A_{(1,3)} \end{bmatrix} \begin{bmatrix} B_{(0,0)} & B_{(0,1)} & B_{(0,2)} \\ B_{(1,0)} & B_{(1,1)} & B_{(1,2)} \\ B_{(2,0)} & B_{(2,1)} & B_{(2,2)} \\ B_{(3,0)} & B_{(3,1)} & B_{(3,2)} \end{bmatrix} \begin{bmatrix} C_{(0,0)} & C_{(0,1)} & C_{(0,2)} \\ C_{(1,0)} & C_{(1,1)} & C_{(1,2)} \end{bmatrix}$$

onde o valor de $C_{(l,c)}$ será calculado como

$$C_{(l,c)} = \sum_{x=0}^{x<A_c} (A_{(l,x)} \times B_{(x,c)})$$

**Exercício 11.106:** ★★★★

Escreva, para a classe `ArrayDePontos2D` (listagem 11.98), um método `menorDistância` que calcule e retorne a menor distância entre dois pontos diferentes que estejam encapsulados no array.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 11.107:** ★★★★

Escreva, para a classe `ArrayDePontos2D` (listagem 11.98), um método `maiorDistância` que calcule e retorne a maior distância entre dois pontos diferentes que estejam encapsulados no array.

**Exercício 11.108:** ★★★★

Um *quadrado mágico* é uma matriz quadrada de valores inteiros onde a soma dos valores em cada linha, coluna ou diagonal principal é a mesma. Por exemplo, a matriz

$$\begin{matrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{matrix}$$

representa um quadrado mágico (onde não existe repetição de valores e todos estão em sequência. Para efeitos deste exercício poderíamos considerar uma matriz onde todos os elementos fossem iguais como sendo também um quadrado mágico).
Escreva uma classe `QuadradoMagico` que tenha o método estático `éQuadradoMágico` que retorne `true` caso a matriz, passada como argumento para o método, represente um quadrado mágico.

**Exercício 11.109:** ★★★★

O jogo japonês *Go* é jogado por duas pessoas em um tabuleiro quadrado de tamanho $19 \times 19$. Cada pessoa recebe um conjunto de peças pretas e brancas que devem ser colocadas alternadamente no tabuleiro, na posição que o jogador desejar. Ganha o jogo o primeiro jogador que conseguir colocar cinco de suas peças em uma linha reta horizontal, vertical ou diagonal.
Crie uma classe em Java que represente um jogo de *Go*, usando uma matriz de duas dimensões para representar as posições do jogo. A matriz deve ser alocada no construtor da classe, ter o tamanho $19 \times 19$ e ser de um tipo que suporte três estados possíveis: vazio, preenchido com peça preta e preenchido com peça branca. A classe deve poder ser usada para jogos com dois jogadores.
A classe deve ter os seguintes métodos:

- `jogaPreta`, que aceita dois valores que são as coordenadas onde uma peça preta será jogada, e marca na matriz a posição **somente** se esta estiver livre.
- `jogaBranca`, que aceita dois valores que são as coordenadas onde uma peça branca será jogada, e marca na matriz a posição **somente** se esta estiver livre.
- `verifica`, que verifica a matriz para ver se existe algum ganhador (este método deve verificar se existem cinco peças iguais que não sejam vazias em uma horizontal, vertical ou diagonal da matriz, depois de **cada** jogada feita).
- `toString`, que retornará uma string com a representação gráfica do jogo com as posições atuais.

Escreva também um programa que use a classe. Este programa deve executar um laço no qual fica perguntando as posições para os jogadores alternadamente, enquanto não houver vitória, desistência ou acabarem as posições vazias da matriz.
*Dica:* O algoritmo do jogo não é tão diferente do jogo da velha (exercício 11.104), exceto pelo método `verifica`. Este método pode, para cada posição do array bidimensional, ver se existem linhas de cinco peças iguais contadas a partir da posição sendo procurada. O único cuidado adicional é garantir que o algoritmo não procurará peças fora do tabuleiro.

**Exercício 11.110:** ⋆⋆⋆⋆

O *crivo de Eratóstenes* é um algoritmo usado para identificar números primos. O algoritmo (apresentado aqui da maneira mais simples) primeiro declara um array de $N$ posições de valores booleanos, todos iguais a `true` (considerando que em princípio qualquer número pode ser primo). O algoritmo, em seguida, marca todos os elementos do array cujos índices são múltiplos de 2 e maiores que o próprio 2 como `false`, indicando que nenhum múltiplo de dois pode ser primo. O algoritmo repete este último procedimento para todos os valores entre 3 e a metade de $N$. Ao final, os índices dos elementos do array que valerem `false` serão valores não-primos, e os que ainda valerem `true` depois da execução do algoritmo serão primos. *Dica:* para entender melhor o algoritmo, rode uma simulação em papel antes.

**Exercício 11.111:** ⋆⋆⋆⋆

O algoritmo de ordenação de bolha (*bubblesort*) serve para colocar os elementos de uma lista em ordem crescente ou decrescente. O algoritmo é como segue:

1. Percorra o array da primeira posição até a penúltima, chamando o elemento do array nesta posição de $P$;
2. Para cada $P$ percorra o mesmo array da próxima posição até a última, chamando o elemento nesta posição de $Q$.
3. Se $P > Q$, troque os valores de $P$ e $Q$.

Uma simulação simples do algoritmo é mostrada abaixo. Consideremos o array $\{7,5,1,3\}$ que deve ser ordenado de forma crescente. Os passos da simulação seriam:

1. O array vale, inicialmente, $\{7,5,1,3\}$.
2. A posição de $P$ no array é 0, e seu valor é 7. A posição de $Q$ no array é 1, e seu valor é 3. Como $P > Q$, os valores são trocados. Agora o array é $\{5,7,1,3\}$.
3. A posição de $P$ no array é 0, e seu valor é 5. A posição de $Q$ no array é 2, e seu valor é 1. Como $P > Q$, os valores são trocados. Agora o array é $\{1,7,5,3\}$.
4. A posição de $P$ no array é 0, e seu valor é 1. A posição de $Q$ no array é 3, e seu valor é 3. Como $P$ não é maior que $Q$, os valores não são trocados. O array permanece sendo $\{1,7,5,3\}$.
5. A posição de $P$ no array é 1, e seu valor é 5. A posição de $Q$ no array é 2, e seu valor é 7. Como $P$ não é maior que $Q$, os valores não são trocados. O array permanece sendo $\{1,7,5,3\}$.
6. A posição de $P$ no array é 1, e seu valor é 5. A posição de $Q$ no array é 3, e seu valor é 3. Como $P > Q$, os valores são trocados. Agora o array é $\{1,3,7,5\}$.
7. A posição de $P$ no array é 2, e seu valor é 7. A posição de $Q$ no array é 3, e seu valor é 5. Como $P > Q$, os valores são trocados. Agora o array é $\{1,3,5,7\}$.
8. O array vale, finalmente, $\{7,5,1,3\}$

Implemente na classe `ArrayDeFloats` (listagem 11.4) o método `ordenaCrescentePorBolha` que implemente o algoritmo descrito. Crie também o método `ordenaDecrescentePorBolha` que implemente um algoritmo que ordene de forma decrescente: a única diferença do algoritmo que ordena de forma crescente é que $P$ e $Q$ deverão ser trocados se $P < Q$.

**Exercício 11.112:** ★★★★

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `parteTriangularSuperior` que retorne uma matriz irregular contendo os valores da matriz (se esta for triangular superior) ou `null` (se a matriz não for triangular superior). Veja também os exercícios 11.56 e 11.76.

**Exercício 11.113:** ★★★★

Escreva, para a classe `MatrizDeDoubles` (listagem 11.9), o método `parteTriangularInferior` que retorne uma matriz irregular contendo os valores da matriz (se esta for triangular inferior) ou `null` (se a matriz não for triangular inferior). Veja também os exercícios 11.56 e 11.76.

**Exercício 11.114:** ★★★★

Um outro algoritmo de ordenação bastante conhecido é o algoritmo de ordenação por seleção (*selection sort*). Este algoritmo funciona de maneira simples: primeiro, cria um array do mesmo tamanho do array a ser ordenado, depois varre sucessivamente o array original procurando o menor valor. Este menor valor é inserido no array de destino, e o valor é removido do array original. O algoritmo repete estes passos até que todos os elementos do array original tenham sido removidos. Como arrays em Java tem tamanho imutável, não é simples remover um elemento de um array de forma que o tamanho do array encolha. Para facilitar a implementação deste algoritmo, usamos o conceito de marcadores (*flags*) - valores que são colocados no lugar do valor removido para indicar que este foi removido.
Uma simulação deste algoritmo que ordena o array $\{41, -72, -25, 25, -3, -85, 19, -63, 44\}$ é mostrada abaixo, (onde os caracteres *** indicam que o elemento já foi ordenado e não deve ser considerado pelo passo, e os mesmos caracteres em azul indicam o elemento que foi retirado por último).

| Passo | Array original | | | | | | | | | Array ordenado |
|---|---|---|---|---|---|---|---|---|---|---|
| início | 41 | -72 | -25 | 25 | -3 | -85 | 19 | -63 | 44 | {} (vazio) |
| 1 | 41 | -72 | -25 | 25 | -3 | *** | 19 | -63 | 44 | $\{-85\}$ |
| 2 | 41 | *** | -25 | 25 | -3 | *** | 19 | -63 | 44 | $\{-85, -72\}$ |
| 3 | 41 | *** | -25 | 25 | -3 | *** | 19 | *** | 44 | $\{-85, -72, -63\}$ |
| 4 | 41 | *** | *** | 25 | -3 | *** | 19 | *** | 44 | $\{-85, -72, -63, -25\}$ |
| 5 | 41 | *** | *** | 25 | *** | *** | 19 | *** | 44 | $\{-85, -72, -63, -25, -3\}$ |
| 6 | 41 | *** | *** | 25 | *** | *** | *** | *** | 44 | $\{-85, -72, -63, -25, -3, 19\}$ |
| 7 | 41 | *** | *** | *** | *** | *** | *** | *** | 44 | $\{-85, -72, -63, -25, -3, 19, 25\}$ |
| 8 | *** | *** | *** | *** | *** | *** | *** | *** | 44 | $\{-85, -72, -63, -25, -3, 19, 25, 41\}$ |
| 9 | *** | *** | *** | *** | *** | *** | *** | *** | *** | $\{-85, -72, -63, -25, -3, 19, 25, 41, 44\}$ |

Não podemos simplesmente usar um valor numérico qualquer como marcador, precisando usar um valor que seja claramente um valor não válido. Se o array for de valores do tipo `float` poderemos usar, então, o valor `Float.NaN`, que não pode ser comparado com nenhum outro valor (bizarramente, a expressão `(Float.NaN == Float.NaN)` retorna `false` e a expressão `(Float.NaN != Float.NaN)` retorna `true` !).
Crie, para a classe `ArrayDeFloats` (listagem 11.4) o método `ordenaCrescentePorSeleção` que implemente o algoritmo descrito. Escreva também o método `ordenaDecrescentePorSeleção` que implemente um algoritmo que ordene de forma decrescente: a única diferença do algoritmo que ordena de forma crescente é que ao invés de procurar o menor valor no array original deveremos procurar o maior array.

**Exercício 11.115:** ★★★★

Considere o método `existe`, pedido como solução do exercício 11.44. Este método é potencialmente ineficiente, pois se o elemento que desejamos encontrar estiver próximo do fim do array, o método deverá percorrer grande parte do array.
Se o array estiver ordenado (veja o exercício 11.111) existe um algoritmo muito mais eficiente de busca, chamado *busca binária*, que funciona através da divisão do array em áreas nas quais a busca pode ser bem-sucedida. Este algoritmo retorna o índice do elemento procurado ou `-1` se o elemento não existir no array, e funciona com os seguintes passos:

1. Iniciamos o algoritmo estabelecendo a área de pesquisa, fazendo com que `primeiro` seja igual a `0` e `último` seja igual ao último índice válido do array.
2. Verificamos se o elemento procurado é igual a `array[primeiro]`. Se for, retornamos `primeiro`.
3. Verificamos se o elemento procurado é igual a `array[último]`. Se for, retornamos `último`.
4. Verificamos a diferença entre `último` e `primeiro` - se for igual a 1, significa que o array sendo procurado somente tem duas posições, mas como o valor procurado não é nenhum dos dois extremos, ele não existe no array, e o algoritmo retorna zero.
5. Calculamos a posição central do array como `primeiro+(último-primeiro)/2` e a armazenamos em `índiceCentral`. Este cálculo deve ser feito usando valores inteiros, pos o índice de um array é sempre inteiro.
6. Se o valor procurado for maior do que `array[índiceCentral]`, chamamos o algoritmo recursivamente, fazendo com que `primeiro` seja igual a `índiceCentral`.
7. Se o valor procurado não for maior do que `array[índiceCentral]`, chamamos o algoritmo recursivamente, fazendo com que `último` seja igual a `índiceCentral`.

Uma simulação do algoritmo, usando o array $\{1,7,8,10,11,13,21,39,41,46,47,50,51,54,58,61,62,67,80,90,96,97,99,100\}$, e procurando neste o valor `80`, é mostrada abaixo:

1. O algoritmo inicia a busca entre os valores 1 e 100, com `primeiro` valendo 0 e `último` valendo 23.
2. Como o valor procurado não é igual a 1 nem igual a 100, calculamos `índiceCentral` como sendo 11. O valor de `array[índiceCentral]` é 50. Como $80 > 50$, chamamos o algoritmo recursivamente com `primeiro` valendo 11 e `último` valendo 23.
3. Como o valor procurado não é igual a 50 nem igual a 100, calculamos `índiceCentral` como sendo 17. O valor de `array[índiceCentral]` é 67. Como $80 > 67$, chamamos o algoritmo recursivamente com `primeiro` valendo 17 e `último` valendo 23.
4. Como o valor procurado não é igual a 67 nem igual a 100, calculamos `índiceCentral` como sendo 20. O valor de `array[índiceCentral]` é 96. Como 80 não é maior que 96, chamamos o algoritmo recursivamente com `primeiro` valendo 17 e `último` valendo 20.
5. Como o valor procurado não é igual a 67 nem igual a 96, calculamos `índiceCentral` como sendo 18. O valor de `array[índiceCentral]` é 80. Como 80 não é maior que 80, chamamos o algoritmo recursivamente com `primeiro` valendo 17 e `último` valendo 18.
6. Como o valor procurado é igual a `80`, retornamos `último`, que é igual a 18.

*Dica:* existem maneiras de otimizar mais ainda este algoritmo. Você consegue imaginar alguma ?

**Exercício 11.116:** ★★★★

Uma das características do algoritmo de ordenação por seleção é a necessidade de marcar valores já selecionados como tendo sido removidos. Na descrição do algoritmo mostrada no exercício 11.114 usamos o valor especial `Float.NaN` para marcar os valores removidos, mas isto impossibilita a ordenação de um array que já contenha um ou mais valores `Float.NaN`.
Uma solução para este problema seria criar um array de valores booleanos associado ao array de valores de ponto flutuante, que representaria, para cada valor, se este já foi removido (`true`) ou não (`false`). Desta forma, qualquer valor presente no array de valores de ponto flutuante poderia ser considerado para ordenação. Escreva versões dos métodos `ordenaCrescentePorSeleção` e `ordenaDecrescentePorSeleção` que usem esta abordagem. *Dica:* como valores `Float.NaN` não podem ser comparados com outros valores, escreva os métodos de forma que se existirem valores `Float.NaN` no array, estes sejam colocados no final do array, mesmo que fiquem após valores `Float.POSITIVE_INFINITY`.

**Exercício 11.117:** ★★★★★

Considere os quadrados mágicos descritos no exercício 11.108. Uma aplicação poderia tentar criar quadrados mágicos de $3 \times 3$ posições com valores inteiros, não repetidos e sequenciais como o mostrado. Um algoritmo de força bruta poderia testar todas as variações dos valores de 1 a 9 colocados em cada uma das nove posições da matriz e usar o método `éQuadradoMágico` da classe `QuadradoMagico` para verificar quais das combinações corresponde a um quadrado mágico, mas o número total de variações é $9^9 = 387420489$ - se cem combinações fossem testadas por segundo, este algoritmo demoraria mais de um mês para testar todas !
Um algoritmo mais eficiente tentaria calcular as variações levando em conta que os valores não devem ser repetidos: se o valor 1 for colocado na posição superior esquerda da matriz, ele não deverá ser usado nas outras posições e assim em diante - desta forma o número de quadrados mágicos a ser testado seria $9! = 362880$ - menos do que um milésimo do valor anterior.
Um algoritmo ainda mais eficiente consideraria que nem todas as combinações de valores devem ser testadas - por exemplo, se o valor 1 está na posição superior esquerda da matriz, sabemos que a soma dos dois valores nas outras colunas da primeira linha devem somar 14, já que a soma da linha deve ser 15. Possivelmente uma solução recursiva poderia ser usada neste algoritmo.
Crie, na classe `QuadradoMagico`, um método `calculaTodos` que calcule todos os quadrados mágicos de tamanho $3 \times 3$ cujos valores sejam consecutivos e não-repetidos entre 1 e 9.

**Exercício 11.118:** ★★★★★

É possível embutir um pouco de *Inteligência Artificial* no jogo da velha do exercício 11.104, fazendo com que um jogador jogue contra o computador. Quando for a vez do computador jogar, as coordenadas onde este colocará sua peça não serão entradas via teclado: a própria classe pode escolher a melhor posição vazia para jogar sua peça baseado na seguinte *heurística* (série de passos que podem levar à solução de um problema): Para cada posição desocupada no tabuleiro, some:

- Mais dois pontos se a posição for a central,
- Mais um ponto se a posição for nos quatro cantos da matriz,
- Menos dois pontos se já houver uma ou mais peças do adversário na linha, coluna ou diagonal onde a posição se encontra,
- Mais quatro pontos se a posição impedir a vitória do adversário,
- Mais quatro pontos se a posição levar a uma vitória,
- Ao final do cálculo, escolher a posição que teve maior número de pontos.

Para exemplificar, considere a figura 11.3, que representa um jogo em andamento, onde o computador joga com as peças `'O'`. No exemplo mostrado, a melhor posição para o computador jogar seria aquela cujo valor é `+2`. As posições marcadas com `não` já estariam ocupadas.
Usando o exercício 11.104 como base, escreva um método `jogaComputador` que calcule a melhor posição para jogo e efetue a jogada. Outras partes da classe deverão ser reescritas, por exemplo, para permitir que o usuário decida se vai jogar com os `'X'` ou `'O'` e quem será o primeiro a jogar.
*Dica:* a classe pode conter outra matriz de valores inteiros, do mesmo tamanho do tabuleiro do jogo da velha, que será reinicializada e calculada com o algoritmo acima a cada jogada do computador.

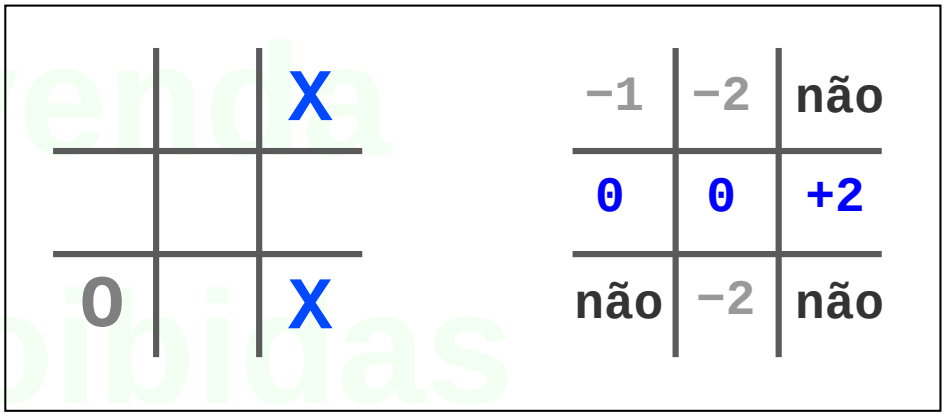



Figura 11.3: Jogo da velha e respectiva matriz de decisão da próxima jogada



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 11.119:** ★★★★★

Também é possível adaptar o jogo de *go* (exercício 11.109) para que o computador possa jogar com o usuário, usando alguma inteligência para decidir onde posicionar suas peças. A heurística para que o computador decida qual posição é melhor para jogar uma peça pode ser:

- Menos dois pontos para cada peça do adversário que estiver na vizinhança direta $7 \times 7$ e mais dois pontos para cada peça do computador que estiver nesta vizinhança,
- Menos quatro pontos para cada peça do adversário que estiver na vizinhança direta $5 \times 5$ e mais quatro pontos para cada peça do computador que estiver nesta vizinhança,
- Menos oito pontos para cada peça do adversário que estiver na vizinhança direta $3 \times 3$ e mais oito pontos para cada peça do computador que estiver nesta vizinhança,
- Menos um ponto para cada peça do adversário que estiver na vizinhança indireta $7 \times 7$ e mais um ponto para cada peça do computador que estiver nesta vizinhança.

A figura 11.4 mostra, para uma determinada posição, quais são as posições que correspondem às vizinhanças.

Usando o exercício 11.109 como base, escreva um método `jogaComputador` que calcule a melhor posição para jogo e efetue a jogada. Outras partes da classe deverão ser reescritas, por exemplo, para permitir que o usuário decida se vai jogar com as peças pretas ou brancas e quem será o primeiro a jogar.

*Dica:* veja também o enunciado do exercício 11.118.

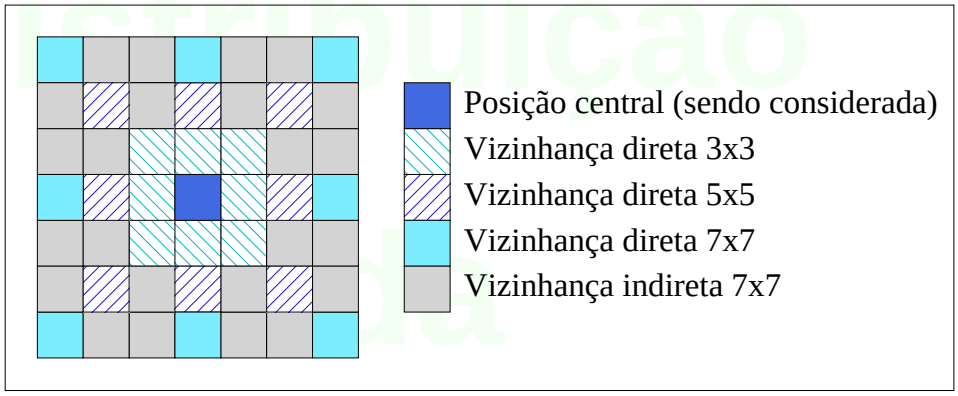



Figura 11.4: Vizinhanças para cálculo da heurística de jogo do jogo *go*

**Exercício 11.120:** ★★★★★

Escreva uma classe `SistemaDeEquacoesLineares` que encapsule simultaneamente uma matriz de $M$ linhas por $N$ colunas e um array unidimensional de $M$ valores de ponto flutuante. Esta classe deverá encapsular um sistema de equações lineares, sendo que a matriz corresponderá aos coeficientes das incógnitas do sistema e o array dos termos independentes. Esta classe deverá ter um construtor, um método para inicializar os valores da matriz e array encapsulados, um método `toString` e um método `resolve` que resolva este sistema, retornando um array com os valores das incógnitas do sistema.
Para resolver o sistema de equações lineares representado por esta classe podemos usar o método de Gauss. Este método consiste em dois algoritmos, um para transformar a matriz de coeficientes em uma matriz triangular superior que lhe seja equivalente, e outro para resolver o sistema de equações por substituições. Ambos os algoritmos são descritos abaixo:

*Parte 1: transformação da matriz de coeficientes das incógnitas em triangular superior*

1. Faça para cada linha $r$ os passos 2 a 5:
2. Marque o elemento na posição $(r,r)$ da matriz como sendo o pivô.
3. Para cada linha cujo índice $w$ seja maior que $r$, calcule um fator de divisão que será o valor negativo da divisão do elemento na linha $w$ e coluna $r$ pelo pivô, ou seja, $-matriz[w][r]/matriz[r][r]$.
4. Divida os elementos da linha $w$ da matriz pelo fator de divisão calculado. Divida também o elemento da linha $w$ do array de termos independentes pelo fator de divisão.
5. Some, na linha $w$, os elementos da linha $r$, elemento a elemento. Some também ao elemento $w$ do array de termos independentes o elemento $r$ do mesmo array.

A figura 11.5 demonstra com um exemplo prático a transformação da matriz de coeficientes em triangular superior.

*Parte 2: solução do sistema usando substituições*

1. Faça para cada linha $r$ os passos 2 a 6, iniciando da última linha e indo para a primeira:
2. Marque o elemento na posição $(r,r)$ da matriz como sendo o pivô.
3. Divida todos os elementos da matriz na linha $r$ pelo pivô. Divida também o elemento da linha $r$ do array de termos independentes pelo pivô.
4. Para cada linha cujo índice $w$ seja menor que $r$, calcule um fator de divisão que será o valor negativo do elemento na linha $w$ e última coluna da linha, ou seja, $-matriz[w][N]$ onde $N$ é o número de colunas da matriz.
5. Divida os elementos da linha $w$ da matriz pelo fator de divisão calculado. Divida também o elemento da linha $w$ do array de termos independentes pelo fator de divisão.
6. Some, na linha $w$, os elementos da linha $r$, elemento a elemento. Some também ao elemento $w$ do array de termos independentes o elemento $r$ do mesmo array.
7. Somente para a primeira linha da matriz, repita os passos 2 e 3.

A figura 11.6 demonstra com um exemplo prático a solução do sistema usando substituições.

Ao final do algoritmo, o array de valores independentes conterá o array de soluções para o sistema de equações.

*Dicas:* a criação desta classe pode ser consideravelmente simplificada se ela for declarada como herdeira da classe `MatrizDeDoubles`, assim os métodos criados como resposta aos exercícios 11.38, 11.39, 11.78 e 11.82 podem ser utilizados.

Eventualmente o sistema de equações lineares não terá solução (se a matriz triangular obtida tiver algum elemento na diagonal principal igual a zero). Neste caso, o método `resolve` deverá retornar `null`.

Esta figura demonstra o primeiro passo do algoritmo da resolução de um sistema de equações lineares pelo método de Gauss (transformação da matriz de coeficientes em triangular superior).

Usaremos o sistema mostrado ao lado como exemplo (onde a matriz de coeficientes é mostrada à esquerda e o array de termos independentes à direita).

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 4.00 & 4.00 & -3.00 \\ 2.00 & -3.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ 3.00 \\ -1.00 \end{bmatrix}$$

Usando o primeiro elemento da primeira linha como pivô calculamos fatores de divisão para a segunda e terceira linhas como sendo o valor negativo dos primeiros elementos destas linhas divididos pelo primeiro elemento da primeira linha. Somente é preciso processar as linhas abaixo da linha do pivô. Desta forma o fator de divisão para a segunda linha será $-4/2 = -2$ e o fator para a terceira linha será $-2/2 = -1$. Dividimos as linhas pelos seus respectivos fatores. A matriz e array resultantes são mostradas ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ -2.00 & -2.00 & 1.50 \\ -2.00 & 3.00 & -1.00 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ -1.50 \\ 1.00 \end{bmatrix}$$

Somamos agora à segunda e terceira linhas os elementos da primeira linha. A matriz e array resultantes são mostrados ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 0.00 & 1.00 & 0.50 \\ 0.00 & 6.00 & -2.00 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ 3.50 \\ 6.00 \end{bmatrix}$$

A primeira coluna já foi processada e todos os seus elementos (exceto o da primeira linha) são iguais a zero. Devemos repetir estes passos considerando como pivô o segundo elemento da segunda coluna. O fator de divisão para a terceira linha (ou seja, o valor do segundo elemento da terceira linha dividido pelo pivô) será $-6/1 = -6$. Dividimos a terceira linha pelo seu fator de divisão obtendo a matriz e array ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 0.00 & 1.00 & 0.50 \\ 0.00 & -1.00 & 0.33 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ 3.50 \\ -1.00 \end{bmatrix}$$

Finalmente, somamos à terceira linha os elementos da segunda linha. A matriz e array são mostradas ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 0.00 & 1.00 & 0.50 \\ 0.00 & 0.00 & 0.83 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ 3.50 \\ 2.50 \end{bmatrix}$$

A segunda parte do algoritmo da resolução de um sistema de equações lineares é mostrado na figura 11.6.

Figura 11.5: Primeiro passo do algoritmo da resolução de um sistema de equações lineares pelo método de Gauss.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Esta figura demonstra o segundo passo do algoritmo da resolução de um sistema de equações lineares pelo método de Gauss (solução do sistema usando substituições). Usemos o exemplo mostrado na figura 11.5 como ponto de partida.

Ao lado, a matriz de coeficientes é mostrada à esquerda e o array de termos independentes à direita.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 0.00 & 1.00 & 0.50 \\ 0.00 & 0.00 & 0.83 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ 3.50 \\ 2.50 \end{bmatrix}$$

Primeiro, dividimos a última linha da matriz e o último elemento do array de termos independentes pelo valor selecionado como pivô (0.83), obtendo a matriz e array mostradas ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 0.00 & 1.00 & 0.50 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ 3.50 \\ 3.00 \end{bmatrix}$$

Calculamos o fator de divisão para a segunda linha, que será o valor negativo do elemento naquela linha e última coluna, ou seja, $-0.50$. Fazemos o mesmo para a primeira linha, obtendo 1.00. Dividimos cada elemento das primeira e segunda linha por seus respectivos fatores de divisão, obtendo a matriz e array mostradas ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & -1.00 \\ 0.00 & -2.00 & -1.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 5.00 \\ -7.00 \\ 3.00 \end{bmatrix}$$

Somamos, nas primeira e segunda linhas, os elementos da terceira linha, elemento a elemento, obtendo a matriz e array mostradas ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & 0.00 \\ 0.00 & -2.00 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 8.00 \\ -4.00 \\ 3.00 \end{bmatrix}$$

Selecionamos como o próximo pivô o elemento da segunda linha que está na diagonal da matriz, ou seja, $-1$. Dividimos a segunda linha da matriz e o elemento correspondente do array de termos independentes pelo valor selecionado como pivô, obtendo a matriz e array mostradas ao lado.

$$\begin{bmatrix} 2.00 & 3.00 & 0.00 \\ 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 8.00 \\ 2.00 \\ 3.00 \end{bmatrix}$$

Calculamos o fator de divisão para a primeira linha, que será o valor negativo do elemento naquela linha e penúltima coluna, ou seja, $-3.00$. Dividimos cada elemento da primeira linha por este fator, obtendo a matriz e array mostradas ao lado.

$$\begin{bmatrix} -0.67 & -1.00 & 0.00 \\ 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} -2.67 \\ 2.00 \\ 3.00 \end{bmatrix}$$

Somamos, na primeira linha, os elementos da segunda linha, elemento a elemento, obtendo a matriz e array mostradas ao lado.

$$\begin{bmatrix} -0.67 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} -0.67 \\ 2.00 \\ 3.00 \end{bmatrix}$$

Finalmente, para a primeira linha, obtemos o pivô como sendo o elemento que está na diagonal ($-0.67$) e dividimos todos os elementos da primeira linha por este valor, obtendo o resultado final mostrado ao lado.

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 \\ 0.00 & 1.00 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{bmatrix} \quad \begin{bmatrix} 1.00 \\ 2.00 \\ 3.00 \end{bmatrix}$$

A matriz de coeficientes se tornou igual à matriz identidade e o array de termos independentes contém os valores da solução para o sistema.

Figura 11.6: Segundo passo do algoritmo da resolução de um sistema de equações lineares pelo método de Gauss.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 11.121:** ★★★★★

O *Jogo da Vida* é um algoritmo de simulação com aplicações e resultados visuais interessantes. Ele simula uma colônia de células em uma grade retangular onde cada elemento da grade pode conter ou não uma célula (veja as figuras 11.7 e 11.8). O algoritmo do jogo é o seguinte:

1. Uma população inicial é gerada, aleatoriamente ou explicitamente indicando-se quais posições contém células ou não.
2. Para cada célula, o número de seus vizinhos (numa vizinhança $3 \times 3$) é calculado.
3. A população é recalculada, de forma que uma célula só sobreviverá se o número de vizinhos dela for exatamente igual a um valor pré-definido.
4. Retorna ao passo 2 para a próxima geração/iteração.

A figura 11.7 mostra 24 iterações do jogo em uma grade de 11 por 11 células, onde a primeira iteração ou situação inicial foi criada explicitamente e o número de vizinhos necessários para a sobrevivência de uma célula é 2. Na implementação do programa que criou as imagens, a grade foi considerada fechada, isto é, considerando que a coluna de número 11 fica à esquerda da coluna de número 1, e de forma similar para as linhas.

A figura 11.8 mostra o número de vizinhos para cada uma das células após a situação inicial e nas duas primeiras iterações. Podemos notar que somente as células que tinham dois vizinhos em uma iteração são duplicadas nas próximas.

Dependendo da configuração da grade (tamanhos horizontal e vertical e se a grade é considerada fechada, com as bordas se tocando ou não) e da população inicial, após várias iterações a colônia de células pode se estabilizar, se extinguir ou ficar em constante modificação.

Escreva uma classe em Java que simule o jogo da vida para várias configurações, e um programa para demonstrar o uso desta classe. Como sugestão, esta classe pode ter os seguintes métodos:

- construtor, que recebe o tamanho da colônia e inicializa uma matriz do tamanho especificado,
- `popula`, que popula a colônia aleatoriamente ou explicitamente com células,
- `itera`, que calcula uma iteração da colônia, calculando o número de vizinhos de cada célula e a nova configuração da população da colônia,
- `imprime`, que imprime a situação atual da colônia, usando caracteres `'*'` para representar células e `'.'` para representar espaços em branco.

*Dicas:* Duas matrizes podem ser usadas para calcular a população: uma de inteiros, para conter o número de vizinhos de cada célula, e uma de booleanos, onde cada valor representa a existência ou não de uma célula na posição. O método `itera` deverá calcular uma matriz a partir da outra.

## 11.6 Exercícios complementares do capítulo 11

**Exercício 11.122:** ★

Escreva na classe `ArrayDeFloats` (listagem 11.4) um método `faixaDeValores` que retorne a diferença entre o maior valor e o menor valor do array encapsulado.

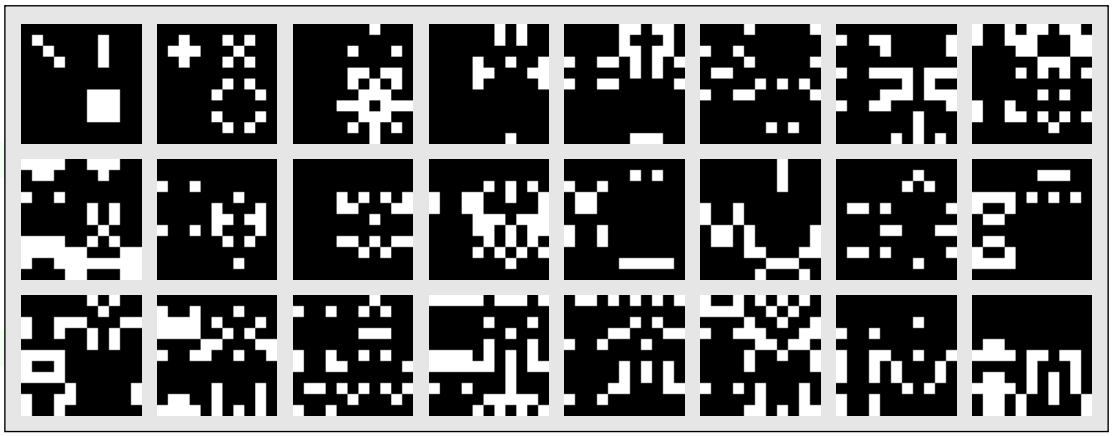


Figura 11.7: Algumas iterações do jogo da vida, mostradas como imagens

Situação Inicial

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | | | | 1 | 1 | 1 | | |
| 1 | 1 | 2 | 1 | | | 2 | 1 | 2 | | |
| 1 | 2 | 2 | 2 | 1 | | 3 | 2 | 3 | | |
| | 1 | 2 | 1 | 1 | | 2 | 1 | 2 | | |
| | | 1 | 1 | 1 | | 1 | 1 | 1 | | |
| | | | | | 1 | 2 | 3 | 2 | 1 | |
| | | | | | 2 | 3 | 5 | 3 | 2 | |
| | | | | | 3 | 5 | 8 | 5 | 3 | |
| | | | | | 2 | 3 | 5 | 3 | 2 | |
| | | | | | 1 | 2 | 3 | 2 | 1 | |

Primeira Iteração

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 1 | | 1 | 1 | 2 | 1 | 1 | |
| 1 | 3 | 3 | 3 | 1 | 1 | 1 | 3 | 1 | 1 | |
| 1 | 3 | 4 | 3 | 1 | 2 | 3 | 4 | 3 | 2 | |
| 1 | 3 | 3 | 3 | 1 | 1 | 1 | 3 | 1 | 1 | |
| | 1 | 1 | 1 | | 2 | 2 | 4 | 2 | 2 | |
| | | | | 1 | 2 | 1 | 2 | 1 | 2 | 1 |
| | | | | 1 | 1 | 2 | 2 | 2 | 1 | 1 |
| | | | | 2 | 2 | 2 | | 2 | 2 | 2 |
| | | | | 1 | 1 | 2 | 2 | 2 | 1 | 1 |
| | | | | 1 | 2 | 1 | 2 | 1 | 2 | 1 |
| | | | | | 1 | 1 | 2 | 1 | 1 | |

Segunda Iteração

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 2 | 1 | 2 | | |
| | | | | 1 | 1 | 2 | 1 | 2 | 1 | 1 |
| | | | | 1 | | 1 | | 1 | | 1 |
| | | | | 2 | 3 | 3 | 2 | 3 | 3 | 2 |
| | | | | 2 | 2 | 3 | 3 | 3 | 2 | 2 |
| | | | | 2 | 3 | 6 | 5 | 6 | 3 | 2 |
| 1 | | | 1 | 3 | 5 | 5 | 5 | 5 | 5 | 3 |
| 1 | | | 1 | 1 | 4 | 5 | 8 | 5 | 4 | 1 |
| 1 | | | 1 | 3 | 5 | 5 | 5 | 5 | 5 | 3 |
| | | | | 1 | 1 | 5 | 4 | 5 | 1 | 1 |
| | | | | 1 | 1 | 4 | 2 | 4 | 1 | 1 |

Figura 11.8: Valores numéricos de algumas iterações do jogo da vida

**Exercício 11.123:** ⋆

Reescreva a classe `CalculoPiQuadradoSobre6` (listagem 11.3) de forma que o valor máximo de iterações seja lido do teclado. Modifique também as chamadas ao método `calculaEMostraSomatória` para que sejam feitas usando argumentos lidos do teclado, para que o usuário possa determinar a precisão desejada. Faça com que a classe verifique os valores passados para que sejam compatíveis com o valor máximo de iterações (evitando, por exemplo, que o usuário peça para calcular a somatória dos primeiros 100000 termos da série, quando o array correspondente só tiver 10000).

**Exercício 11.124:** ⋆

O construtor da classe `JogoDeDamas` (listagem 11.10) inicializa todo o array de duas dimensões com o caracter ponto (`'.'`) para depois reinicializar algumas posições com os caracteres que representam as peças. Modifique o construtor para que este seja mais eficiente, inicializando com pontos somente as posições que não serão inicializadas posteriormente.

**Exercício 11.125:** ⋆

Considere o método `faixaDeValores`, que deve ter sido escrito na classe `ArrayDeFloats` (listagem 11.4) de acordo com o pedido no exercício 11.122. Existem duas abordagens diferentes para criar este método: usar valores que podem ser calculados com métodos já existentes e reescrever o método para que este calcule independentemente a faixa. Existe diferença de performance entre as duas abordagens ? Responda com uma demonstração que verifique a diferença do tempo de execução das duas versões deste método.

**Exercício 11.126:** ⋆

O que aconteceria se o laço no método `menorValor` e `maiorValor` na classe `ArrayDeFloats` (listagem 11.4) iniciássem em zero ao invés de um ? Explique.

**Exercício 11.127:** ⋆

Quantos elementos estão contidos no array irregular mostrado na classe `TrianguloDePascal` (listagem 11.11) ? Quantos seriam usados se utilizássemos um array regular ao invés de irregular ? E se o tamanho do array fosse 1000000 linhas ao invés de 10 ?

**Exercício 11.128:** ⋆⋆

Modifique o método `toString` da classe `JogoDeDamas` (listagem 11.10) para que este imprima uma borda ao redor do tabuleiro. Esta borda pode ser feita com os caracteres `'-'` (para linhas horizontais), `'|'` (para linhas horizontais) e `'+'` para intersecções.

**Exercício 11.129:** ⋆⋆

Considere que as linhas `Funcionario[] equipe = new Funcionario[5]; equipe[5] = new Funcionario();` estejam escritas em um método `main` de uma classe qualquer. Quais das opções abaixo serão verdadeiras ?

**A.** A referência `equipe[5]` valerá `null`.
**B.** A classe não pode ser compilada pois o índice do array não é compatível com o tamanho declarado.
**C.** O campo que representa a data de admissão da referência `equipe[0]` valerá `null`.
**D.** A classe não pode ser compilada pois não existe construtor para a classe `Funcionario` que não receba argumentos.
**E.** Um erro de execução ocorrerá nesta linha pois o índice do array não é compatível com o tamanho declarado.

**Exercício 11.130:** ⋆⋆

Explique, com suas palavras, porque o algoritmo de busca binária somente funciona em arrays ordenados.

**Exercício 11.131:** ⋆⋆⋆

Modifique o método `toString` da classe `JogoDeDamas` (listagem 11.10) para que este imprima um tabuleiro ampliado, de forma que ao invés de um caracter para representar cada peça, quatro caracteres arranjados dois a dois sejam usados. Desta forma, os caracteres `'.'` (ponto, representando posição sem peça), `'o'` e `'x'` (representando as peças dos jogadores) podem ser representados pelos grupos de quatro caracteres mostrados abaixo:

```
..    /\    \/
..    \/    /\
```

Escreva o método de forma que uma borda seja colocada entre cada caracter. Uma amostra de como o tabuleiro deve ser impresso é mostrada abaixo (somente as quatro primeiras colunas das duas primeiras linhas são mostradas):

```
+--+--+--+--+
|/\|..|\/|..|
|\/|..|/\|..|
+--+--+--+--+
|/\|..|..|\/|
|\/|..|..|/\|
+--+--+--+--+
```

*Dica:* para cada linha do array encapsulado, três linhas devem ser impressas, e para cada coluna, três caracteres devem ser impressos.

**Exercício 11.132:** ⋆⋆⋆

Modifique os métodos `menorValor` e `maiorValor` da classe `ArrayDeFloats` (listagem 11.4) para que sejam chamados de forma recursiva.

**Exercício 11.133:** ⋆⋆⋆

Os elementos do array encapsulado pela classe `ArrayDeObjetosGeometricos` (listagem 11.7) são cópias das referências criadas pela classe que usar instâncias de `ArrayDeObjetosGeometricos`, e podem ser modificadas depois de inseridas. Uma solução para evitar isto é garantir que os objetos inseridos no array são clones dos objetos passados como argumento para o método `modifica`. Modifique o método `modifica` para que este use clones dos objetos quando for inserí-los no array. Veja também o exercício 9.28.

# Capítulo 12

# Classes para Manipulação de Strings

## 12.1 Introdução

Em diversas classes vistas até agora, usamos instâncias da classe `String` para representar cadeias de caracteres. Cadeias de caracteres podem ser utilizadas para representação e processamento de qualquer informação textual, o que é frequentemente necessário para aplicações diversas, em especial envolvendo processamento de dados em arquivos e aplicações em rede.

Java tem classes poderosas e flexíveis para o processamento de strings, cada uma com características especiais e métodos úteis, que serão apresentadas neste capítulo.

## 12.2 A classe `String`

*classe* `String`

A classe mais usada para representação e processamento de cadeias de caracteres é a classe `String`, da qual vimos muitos exemplos de uso. Além de permitir a representação de cadeias de caracteres de forma simples, a classe `String` contém muitos métodos práticos e úteis para o processamento de textos.

*composição de strings*

Um conceito importante sobre strings é que instâncias desta classe são compostas de zero ou mais caracteres (valores do tipo char) enfileirados em ordem de leitura (esquerda para direita). Por exemplo, a string `"Java"` é composta dos caracteres `'J'`, `'a'`, `'v'` e `'a'`, sendo que `'J'` é considerado o primeiro caracter da string e `'a'` o último. O *comprimento* da string é dado pelo número de caracteres que esta contém: no exemplo anterior, a string contém quatro caracteres.

*comprimento de strings*

*índice de caracteres em strings*

O *índice* de cada um dos caracteres indica a posição deste na string: este índice pode valer entre zero e o valor do comprimento da string menos um. Por exemplo, a posição do caracter `'J'` na string `"Java"` é zero, e do caracter `'v'` na mesma string é dois. Os índices na string podem estar entre zero e três. Tentativas de acessar caracteres ou trechos de uma string usando índices fora da faixa de valores válidos causarão uma exceção chamada `java.lang.StringIndexOutOfBoundsException` (exceções são categorias especiais de erros em Java, e serão vistas com detalhes no capítulo **??**).

### 12.2.1 Construindo strings

Nos exemplos vistos anteriormente vimos que instâncias da classe `String` podem ser construídas diretamente, por atribuição direta à referências, sem necessidade da chamada do construtor atrav, por atribuição direta à referências, sem necessidade da chamada do construtor através da



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

palavra-chave `new` - isto é uma exceção entre as outras classes que vimos.

*construtores para a classe* `String`

Instâncias da classe `String` também podem ser construídas de forma mais tradicional, usando a palavra-chave `new` e um construtor que recebe, como argumento, uma outra instância, já existente, da classe `String`. O trecho de código `String ciência = new String("Astronomia");` resultará em uma nova instância da classe `String` contendo a cadeia de caracteres `"Astronomia"`, sendo, para finalidades práticas, idêntico ao trecho `String ciência = "Astronomia";`.

Instâncias da classe `String` também podem ser construídas se passarmos para o construtor da classe um array de caracteres que já tenha sido inicializado de alguma forma. Assim, o trecho de código `char[] caracteres = {'H','a','w','k','i','n','g'}; String cientista = new String(caracteres);` criará uma instância da classe `String` contendo a cadeia de caracteres `"Hawking"`. Não é possível criar uma instância da classe `String` passando como argumento outros tipos de arrays ou dados.

### 12.2.2 Métodos básicos da classe `String`

*método* `length`

Alguns métodos básicos da classe `String` servem para que possamos processar os caracteres que formam a string. Um destes métodos é o método `length`, que não recebe argumentos e retorna um valor inteiro igual ao número de caracteres contidos na string. Este método retorna o número total de caracteres na string, e não somente os caracteres visíveis - caracteres de controle (veja a tabela A.1) também são contados. É importante notar que o método `length` da classe `String` é diferente do campo `length` de arrays em Java - a sintaxe de chamada de métodos exige que usemos parênteses, mesmo que argumentos não sejam passados.

*método* `charAt`

Um outro método da classe `String` permite que recuperemos caracteres individuais da string. Este método é o `charAt`, que recebe como argumento um valor inteiro correspondente à posição do caracter que desejamos recuperar, e retorna o caracter naquela posição. Usando os métodos `length` e `charAt` podemos facilmente escrever um laço que recupere todos os caracteres da string, um a um.

Para demonstrar usos dos métodos `length` e `charAt` consideremos um jogo simples de Forca, onde o objetivo é adivinhar uma palavra oculta verificando que caracteres aparecem em que posições nesta palavra. Neste jogo, um jogador escolherá uma letra do alfabeto de cada vez, analisando a posição na palavra oculta onde esta letra aparece (se aparecer).

O algoritmo simplificado do jogo pode ser implementado com os seguintes passos:

1. O algoritmo recebe a palavra a ser adivinhada, e cria um array de valores booleanos, um para cada caracter da palavra, inicializados com `false`, significando que o caracter não foi adivinhado ainda. O algoritmo também declara e inicializa um array de 26 posições booleanas, correspondentes às letras do alfabeto que já foram utilizadas na adivinhação, também inicializados com `false`.
2. O algoritmo então inicia a interação com o usuário, mostrando a estes quais letras já foram adivinhadas e quais já foram utilizadas.
3. O algoritmo então pergunta ao usuário uma letra, marcando esta letra como usada e marcando as letras da palavra iguais a esta como já adivinhadas.
4. Se todas as letras da palavra tiverem sido adivinhadas, encerra o jogo, caso contrário volta ao passo 2. Este algoritmo simplificado não considera um número máximo de tentativas por parte do jogador.

A classe JogoDaForca, mostrada na listagem 12.1, contém campos para representar a palavra e arrays mostradas no algoritmo acima, assim como métodos para implementar os passos do algoritmo.

Listagem 12.1: A classe JogoDaForca, que encapsula os mecanismos de um jogo da forca simples.

```
1  /**
2   * A classe JogoDaForca, que encapsula os mecanismos de um jogo da forca simples. O
3   * objetivo do jogo é adivinhar a palavra determinada por um adversário em um número
4   * mínimo de tentativas. A classe implementa o construtor, um método que permite a
5   * interação com o usuário, métodos para a apresentação de resultados parciais e um
6   * método que verifica se a palavra já foi completamente adivinhada.
7   * Esta classe assume que os caracteres da palavra encapsulada e os caracteres
8   * entrados pelo método pergunta são todos minúsculos.
9   */
10 class JogoDaForca // declaração da classe
11   {
12   /**
13    * Os campos da classe.
14    */
15    private String palavra; // a palavra encapsulada pelo jogo, que deve ser adivinhada
16    private int comprimento; // o comprimento desta palavra, em caracteres
17    private boolean[] corretas; // um array de valores booleanos que indicará, para
18                                // cada letra, se ela já foi corretamente adivinhada
19    private boolean[] letrasUtilizadas; // um array de valores booleanos que indicará,
20                                        // para cada letra do alfabeto, se ela já foi
21                                        // utilizada
22
23   /**
24    * O construtor da classe. O constutor receberá um argumento que é a palavra que deve
25    * ser adivinhada, e populará os outros campos da classe de acordo com esta palavra.
26    * @param aPalavra a palavra a ser adivinhada
27    */
28    JogoDaForca(String aPalavra)
29      {
30      // Inicializo a palavra a ser adivinhada.
31      palavra = aPalavra;
32      // Calculo e armazeno seu comprimento, isto é, seu número de caracteres.
33      comprimento = palavra.length();
34      // Crio um array com um valor booleano para cada caracter na palavra.
35      corretas = new boolean[comprimento];
36      // Crio um array com um valor booleano para cada letra do alfabeto.
37      letrasUtilizadas = new boolean[26];
38      }
39
40   /**
41    * O método pergunta, que mostra o resultado do jogo até o presente momento e
42    * pergunta ao usuário qual é o próximo caracter a ser adivinhado. Este método
43    * modifica os arrays encapsulados pela classe, atualizando as letras da palavra que
44    * já foram adivinhadas corretamente e as letras do alfabeto que já foram utilizadas.
45    */
46    public void pergunta()
47      {
48      // Mostra ao jogador quais letras da palavra já foram adivinhadas
49      mostraAdivinhadas();
50      // Mostra ao jogador quais letras do alfabeto já foram utilizadas
51      mostraUtilizadas();
52      // Peço ao usuário que entre um caracter
53      System.out.print("Entre um caracter:");
54      char tentativa = Keyboard.readChar();
55      // Dizemos que aquela letra já foi utilizada
56      letrasUtilizadas[tentativa-'a'] = true;
57      // Marcamos todas as letras iguais a esta no array de letras ocultas
58      for(int contador=0;contador<comprimento;contador++)
59        {
60        if (palavra.charAt(contador) == tentativa)
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
61          corretas[contador] = true;
62        }
63      }
64
65    /**
66     * O método mostraAdivinhadas, que mostra quais letras da palavra já foram
67     * adivinhadas. Este método é chamado como uma subrotina de outro método desta
68     * classe, podendo ser declarado como privado.
69     */
70    private void mostraAdivinhadas()
71      {
72      // Mostro ao usuário as letras da palavra que já foram adivinhados
73      System.out.print("Adivinhado até agora: ");
74      // Para cada letra na palavra...
75      for(int contador=0;contador<comprimento;contador++)
76        {
77        if (corretas[contador])  // se ela já foi adivinhada
78          System.out.print(palavra.charAt(contador)); // imprimo a letra
79        else // senão
80          System.out.print("_"); // imprimo o caracter '_'
81        }
82      System.out.println(); // pulamos uma linha
83      }
84
85    /**
86     * O método mostraUtilizadas, que mostra quais letras do alfabeto já foram
87     * utilizadas. Este método é chamado como uma subrotina de outro método desta
88     * classe, podendo ser declarado como privado.
89     */
90    private void mostraUtilizadas()
91      {
92      // Mostro ao usuários as letras do alfabeto que já foram utilizadas
93      System.out.print("Letras já utilizadas: ");
94      // Para cada letra do alfabeto...
95      for(int contador=0;contador<26;contador++)
96        if (letrasUtilizadas[contador]) // se já foi utilizada
97          System.out.print((char)('a'+contador)); // imprimo a letra
98      System.out.println(); // pulamos uma linha
99      }
100
101   /**
102    * O método quantasLetrasAindaNãoAdivinhadas, que calcula e retorna o número de
103    * letras da palavra que ainda não foram adivinhadas.
104    * @return o número de letras da palavra que ainda não foram adivinhadas
105    */
106   public int quantasLetrasAindaNãoAdivinhadas()
107     {
108     // O total de letras não adivinhadas é inicialmente zero
109     int quantasNãoAdivinhadas = 0;
110     // Para cada letra da palavra...
111     for(int contador=0;contador<comprimento;contador++)
112       if (!corretas[contador])  // se a letra não tiver sido adivinhada ...
113         quantasNãoAdivinhadas++; // incrementa o contador.
114     return quantasNãoAdivinhadas;
115     }
116
117   } // fim da classe JogoDaForca
```

*operações com caracteres*

Alguns truques de manipulação de caracteres são usados na classe `JogoDaForca` para simplificar a indexação do array de caracteres utilizados: considerando que somente caracteres minúsculos serão utilizados, podemos transformar um valor que esteja entre `'a'` e `'z'` em índices para o array entre `0` e `25` subtraindo-se `'a'` do caracter entrado - como caracteres podem ser processados como valores inteiros, a operação é válida. Similarmente, para transformar um valor entre `0` e `25` para um caracter entre `'a'` e `'z'` basta somarmos `'a'` ao valor e fazer um *cast* do valor resultante para o tipo `char`.

A classe DemoJogoDaForca, mostrada na listagem 12.2, demonstra o uso de uma instância da classe JogoDaForca. O algoritmo do jogo foi implementado de forma simples, com a palavra sendo constante e não existindo limite no número de tentativas que o usuário tem para adivinhar a palavra.

Listagem 12.2: A classe DemoJogoDaForca, que demonstra o uso da classe JogoDaForca.

```
1  /**
2   * A classe DemoJogoDaForca, que demonstra o uso da classe JogoDaForca.
3   */
4  class DemoJogoDaForca // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém a declaração
8    * de uma instância da classe JogoDaForca, e executa métodos desta classe
9    * que implementam um jogo simples de Forca.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     // Criamos uma instância da classe, passando para o construtor a palavra
16     // a ser adivinhada
17     JogoDaForca jf = new JogoDaForca("miraculosamente");
18     // Enquanto o jogador não acerta a palavra (isto é, enquanto ainda existem
19     // letras que não foram adivinhadas)...
20     while(jf.quantasLetrasAindaNãoAdivinhadas() > 0)
21       {
22       jf.pergunta(); // perguntamos a nova letra
23       }
24     } // fim do método main
25
26   } // fim da classe DemoJogoDaForca
```

Um outro método da classe String para manipulação dos caracteres da string é o método toCharArray, que retorna um array unidimensional de caracteres criado com o tamanho adequado para receber todos os caracteres da string. Em cada posição do array estará o caracter correspondente da string.

*método* toCharArray

### 12.2.3 Métodos para comparação de Strings

É frequente a necessidade de comparação de valores de variáveis ou campos com outras variáveis, campos ou constantes, para tomada de decisões em programas e métodos. Conforme comentado na seção 2.3.1, instâncias da classe String (sejam elas campos ou mesmo constantes) não podem ser comparadas com o operador ==[1].

*comparação de strings*

Para a comparação de strings podemos usar métodos da própria classe String. Uma vantagem do uso de métodos para comparação de strings é que existem mais de uma maneira de compararmos strings, a mais comum sendo a comparação de caracter por caracter de duas strings, com diferenciação ou não de caracteres maiúsculos e minúsculos. Se considerarmos que caracteres maiúsculos e minúsculos são iguais, as strings "Bioquímica" e "BIOQUÍMICA" serão iguais,

[1] Na verdade, em condições especiais, strings podem ser comparadas com o operador == e os resultados aparecerão corretos. Por exemplo, considerando o trecho de código String s1 = "Apoptose"; String s2 = "Apoptose"; a expressão (s1 == s2) seria igual a true, mas somente porque as duas referências a instâncias da classe String são iguais - o compilador Java simplesmente evita alocar memória duas vezes para o mesmo conteúdo, de forma que o trecho de código mostrado seja compilado como String s1 = "Apoptose"; String s2 = s1;. Por outro lado, considerando o trecho de código String s1 = "Apop"; String s2 = s1+"tose"; a expressão (s2 == "Apoptose") seria igual a false - existem limites na capacidade do compilador de organizar referências para instâncias duplicadas.

mas se considerarmos que caracteres maiúsculos e minúsculos são diferentes, as strings serão diferentes.

*método* `equals`

Um dos métodos básicos de comparação de strings é o método `equals`, que deve ser chamado a partir de uma instância da classe `String` e receber outra instância da classe `String` como argumento, e retornará o valor booleano `true` se todos os caracteres da instância que chamou o método forem iguais (inclusive na ordem e considerando maiúsculos diferentes de minúsculos) aos caracteres da instância passada como argumento. O segundo método básico de comparação de strings é o método `equalsIgnoreCase`, que se comporta da mesma forma que o método `equals`, exceto que considera caracteres maiúsculos e minúsculos como sendo iguais.

*método* `equalsIgnoreCase`

Estes métodos de comparação retornarão o mesmo valor independente da ordem das strings sendo comparadas, isto é, de qual string executará o método e de qual será o argumento para o método. Desta forma, se ambas as strings `nome` e `cientista` contiverem `"Einstein"`, as expressões `nome.equals(cientista)` e `cientista.equals(nome)` serão avaliadas como `true`. Como strings constantes (implementadas como instâncias sem referências associadas) também são instâncias da classe `String`, podemos até mesmo executar métodos nestas constantes, assim `"Einstein".equals(cientista)` e `"einstein".equalsIgnoreCase(nome)` também serão avaliadas como `true`.

A classe `Login` demonstra de forma simples o uso dos métodos `equals` e `equalsIgnoreCase`. Esta classe encapsula um nome e senha, cada um sendo uma instância da classe `String`, e contém métodos para verificar se nome e senha passados como argumentos são iguais aos encapsulados, de forma rigorosa (considerando maiúsculas e minúsculas como sendo diferentes) ou menos rigorosa (considerando maiúsculas e minúsculas como sendo iguais). A classe `Login` é mostrada na listagem 12.3.

Listagem 12.3: A classe `Login`, que encapsula um nome de usuário e senha e um mecanismo para verificar este nome e senha.

```
1  /**
2   * A classe Login, que encapsula um nome de usuário e senha e um mecanismo para
3   * verificar este nome e senha.
4   */
5  class Login // declaração da classe
6    {
7   /**
8    * Os campos da classe, que armazenarão o nome de usuário e a sua senha.
9    * Um campo adicional indicará se a verificação deve ser rigorosa ou não.
10   */
11   private String nome,senha;
12   private boolean verificaçãoRigorosa;
13
14  /**
15   * Um construtor da classe, que recebe como argumentos um nome e uma senha.
16   * Este construtor chama o construtor mais completo, passando para este os
17   * argumentos. Se este construtor for utilizado, consideraremos que a verificação
18   * da senha e nome deve ser rigorosa.
19   * @param nome o nome a ser encapsulado
20   * @param senha a senha a ser encapsulada
21   */
22   Login(String nome,String senha)
23     {
24     this(nome,senha,true); // chama o construtor com mais argumentos
25     }
26
27  /**
28   * Um construtor da classe, que recebe como argumentos um nome, uma senha e
```

```
29   * um valor que indica se a verificação será rigorosa ou não. Este construtor
30   * inicializa os campos da classe com estes argumentos.
31   * @param nome o nome a ser encapsulado
32   * @param senha a senha a ser encapsulada
33   * @param verificaçãoRigorosa indica se a verificação de nome e senha será rigorosa
34   *        ou não.
35   */
36   Login(String nome,String senha,boolean verificaçãoRigorosa)
37     {
38     this.nome = nome;     // inicializa os campos usando os valores passados como
39     this.senha = senha;   // argumentos
40     this.verificaçãoRigorosa = verificaçãoRigorosa;
41     }
42
43  /**
44   * O método verifica, que compara o nome e a senha com valores passados como
45   * argumento, retornando true se o nome e senha encapsulados forem iguais ao nome
46   * e senha passados como argumento.
47   * @param nome o nome a ser comparado com o encapsulado
48   * @param senha a senha a ser comparada com a encapsulada
49   * @return true se o nome e senha forem iguais, false caso contrário
50   */
51   public boolean verifica(String nome,String senha)
52     {
53     // Se a verificação deve ser rigorosa, consideraremos maiúsculas e minúsculas
54     // como sendo diferentes.
55     if (verificaçãoRigorosa)
56       {
57       if (this.nome.equals(nome) &&  // se o nome for igual ao encapsulado
58           this.senha.equals(senha))  // e a senha também for
59           return true; // retornamos true (verificação OK)
60       else return false; // senão retornamos false (verificação falhou)
61       }
62     else // senão, consideraremos maiúsculas e minúsculas como sendo iguais.
63       {
64       if (this.nome.equalsIgnoreCase(nome) &&  // se o nome for igual ao encapsulado
65           this.senha.equalsIgnoreCase(senha))  // e a senha também for
66           return true; // retornamos true (verificação OK)
67       else return false; // senão retornamos false (verificação falhou)
68       }
69     }
70
71   } // fim da classe Login
```

Dois pontos interessantes da classe `Login` (listagem 12.3) são seus construtores, criados usando o conceito de sobrecarga de métodos, e a forma na qual os campos da classe são comparados com os argumentos passados para os métodos: `this.nome.equals(nome)` indica que estaremos executando o método `equals` do campo `nome` (instância da classe `String`) da própria classe (`this`), passando como argumento a referência `nome` que foi passada como argumento para o método `verifica`.

A classe `DemoLogin`, mostrada na listagem 12.4, demonstra o uso de uma instância da classe `Login`. O método `main` desta classe pede a um usuário que digite um nome e senha, que se estiverem corretos (usando o mecanismo de verificação rigorosa da classe `Login`), permitirá ao usuário escolher um autor (entrando o seu nome pelo teclado) para que uma frase famosa deste autor seja mostrada no terminal.

Listagem 12.4: A classe `DemoLogin`, que demonstra o uso de uma instância da classe `Login`.

```
1  /**
2   * A classe DemoLogin, que demonstra o uso de uma instância da classe Login. Esta
3   * classe pede que o usuário se identifique com o nome e senha, e caso estes sejam
4   * iguais aos cadastrados, mostramos uma frase de um autor (também decidido pelo
```

```
5   * usuário).
6   */
7  class DemoLogin // declaração da classe
8    {
9   /**
10   * O método main permite a execução desta classe. Este método contém a declaração
11   * de uma instância da classe Login, e a usa para executar parte do código somente
12   * se o usuário entrar corretamente o nome e a senha.
13   * @param argumentos os argumentos que podem ser passados para o método via linha
14   *        de comando, mas que neste caso serão ignorados.
15   */
16    public static void main(String[] argumentos)
17      {
18      // Criamos uma instância da classe Login, contendo o nome e senha do usuário
19      // autorizado.
20      Login autorizado = new Login("Auberon","Titania");
21      // Perguntamos ao usuário qual é o nome e senha.
22      String nome,senha;
23      System.out.println("Para prosseguir é necessário um nome e senha.");
24      System.out.print("Nome :");
25      nome = Keyboard.readString();
26      System.out.print("Senha:");
27      senha = Keyboard.readString();
28      // Verificamos se o nome e senha são iguais aos passados para o construtor da
29      // classe Login.
30      if (!autorizado.verifica(nome,senha))
31        {
32        System.out.println("O nome e/ou senha fornecidos são diferentes "+
33                           "dos autorizados.");
34        System.exit(1);
35        }
36      // Se chegamos a este ponto, o nome e senha foram reconhecidos. Podemos prosseguir
37      // com a interação com o usuário. Mostramos uma mensagem pedindo que o usuário
38      // entre com o nome do autor da mensagem.
39      System.out.println("Entre o nome do autor da frase. Opções válidas são:");
40      System.out.println("Asimov, Joyce, Knuth, Olsen, Minsky ou Desconhecido.");
41      String autor = Keyboard.readString();
42      if (autor.equalsIgnoreCase("Asimov"))
43        {
44        System.out.println("Eu não tenho medo de computadores. Eu tenho medo da ");
45        System.out.println("falta de computadores.");
46        System.out.println(" -- Isaac Asimov");
47        } // foi igual a Asimov
48      else if (autor.equalsIgnoreCase("Joyce"))
49        {
50        System.out.println("A má reputação que Unix tem é totalmente desmerecida, ");
51        System.out.println("criada por pessoas que não entendem, que não foram lá e ");
52        System.out.println("tentaram alguma coisa.");
53        System.out.println(" -- Jim Joyce, ex-professor de ciência da computação na ");
54        System.out.println("    Universidade da Califórnia");
55        } // foi igual a Joyce
56      else if (autor.equalsIgnoreCase("Knuth"))
57        {
58        System.out.println("Uma pessoa que está mais que superficialmente interessada ");
59        System.out.println("em computadores deveria conhecer bem linguagem de ");
60        System.out.println("máquina, já que isto é parte fundamental de um computador.");
61        System.out.println(" --  Donald Knuth");
62        } // foi igual a Knuth
63      else if (autor.equalsIgnoreCase("Olsen"))
64        {
65        System.out.println("Não existe nenhuma razão para que um indivíduo tenha um ");
66        System.out.println("computador em sua casa.");
67        System.out.println(" -- Ken Olsen (presidente da Digital Equipment ");
68        System.out.println("    Corporation), na convenção da Sociedade do Mundo ");
69        System.out.println("    Futuro, em 1977.");
70        } // foi igual a Olsen
71      else if (autor.equalsIgnoreCase("Minsky"))
72        {
73        System.out.println("Em três a oito anos nós teremos uma máquina com a ");
```

```
74        System.out.println("inteligência de um ser humano médio ... a máquina irá ");
75        System.out.println("começar a se educar com uma velocidade fantástica ... em ");
76        System.out.println("alguns meses ela estará no nível de gênios e alguns meses ");
77        System.out.println("depois seus poderes serão incalculáveis. ");
78        System.out.println(" -- Marvin Minsky, 1970");
79        } // foi igual a Minsky
80      else if (autor.equalsIgnoreCase("Desconhecido"))
81        {
82        System.out.println("A criação de software é sempre um processo exigente e ");
83        System.out.println("esmerado - um exercício em lógica, expressão clara e ");
84        System.out.println("atenção quase fanática para detalhes. Isso requer ");
85        System.out.println("inteligência, dedicação e uma enorme quantidade de ");
86        System.out.println("trabalho duro. Mas, uma certa quantidade de inspiração ");
87        System.out.println("imprevisível e frequentemente não repetitível é o que ");
88        System.out.println("usualmente faz a diferença entre adequação e excelência.");
89        System.out.println(" -- Desconhecido");
90        } // foi igual a Desconhecido
91      else // nenhum dos nomes conhecidos foi selecionado
92        {
93        System.out.println("Você escolheu um nome não reconhecido.");
94        }
95    } // fim do método main
96
97  } // fim da classe DemoLogin
```

Além dos métodos `equals` e `equalsIgnoreCase`, temos mais dois métodos para comparação de strings que consideram somente os caracteres iniciais ou finais destas. O método `startsWith` retorna o valor booleano `true` se os primeiros caracteres da string que executar o método forem iguais aos caracteres da string passada como argumento. Por exemplo, se a string `fabricante` for igual a "`Nikon`" a expressão `fabricante.startsWith("Nik")` séra igual a `true`. Os caracteres maiúsculos são considerados diferentes dos minúsculos, e se o comprimento da string passada como argumento for maior que o da string que chamou o método, este retornará `false`, desta forma as expressões `fabricante.startsWith("Can")` e `fabricante.startsWith("Nikonos")` serão iguais a `false`.

*método* `startsWith`

O método `endsWith` funciona como o método `startsWith`, mas compara somente os caracteres da string passada como argumento com os últimos caracteres da string que executa o método, considerando caracteres maiúsculos e minúsculos como sendo diferentes. Por exemplo, se a string `ciência` for igual a "`Metafísica`", a expressão `ciência.endsWith("física")` será igual a `true`, mas `ciência.endsWith("Física")` e `ciência.endsWith("fisica")` serão iguais a `false`, uma porque terá uma diferença entre um caracter maiúsculo e minúsculo e outra porque terá uma diferença entre um caracter com e outro sem acento.

*método* `endsWith`

A classe `URL` demonstra usos dos métodos `startsWith` e `endsWith`, encapsulando uma string e comparando seus prefixos e sufixos com extensões e protocolos, respectivamente[2]. A classe `URL` é mostrada na listagem 12.5.

Listagem 12.5: A classe `URL`, que representa um URL (*Uniform Resource Locator*), que representa a localização de um recurso na Internet.

```
1 /**
2  * A classe URL, que representa um URL (Uniform Resource Locator), que representa a
3  * localização de um recurso na Internet. Esta classe contém o construtor e alguns
4  * métodos que são úteis para analisar URLs.
5  */
6 class URL // declaração da classe
```

[2] A classe `URL` mostrada neste capítulo somente contém métodos simples para análise da string encapsulada. A classe `URL`, padrão de Java, com funcionalidade real, será vista na seção **??**.

```
7   {
8  /**
9   * O campo da classe, uma única string que armazena a localização do recurso.
10  */
11  private String localização;
12
13 /**
14  * O construtor da classe, que recebe como argumento uma string contendo a
15  * localização do recurso e usa este argumento para inicializar o campo da classe.
16  * @param localização a localização do recurso na Internet
17  */
18  URL(String localização)
19    {
20    this.localização = localização;
21    }
22
23 /**
24  * O método éImagem, que retorna o valor booleano true se o URL apontar para uma
25  * imagem estática. A avaliação é feita considerando-se o sufixo do URL.
26  * @return true se o URL apontar para uma imagem, false caso contrário
27  */
28  public boolean éImagem()
29    {
30    if (localização.endsWith(".gif") || // se o URL terminar com uma destas
31        localização.endsWith(".jpg") || // extensões (maiúsculas ou minúsculas)
32        localização.endsWith(".png") || // consideraremos que ele aponta para uma
33        localização.endsWith(".GIF") || // imagem
34        localização.endsWith(".JPG") ||
35        localização.endsWith(".PNG"))
36      return true;
37    else return false;
38    }
39
40 /**
41  * O método éHTTP, que retorna o valor booleano true se o protocolo do URL for http
42  * (seguro ou não). A avaliação é feita considerando-se o prefixo do URL.
43  * @return true se o protocolo do URL for http, false caso contrário
44  */
45  public boolean éHTTP()
46    {
47    if (localização.startsWith("http://") ||  // se o URL começar com uma destas
48        localização.startsWith("HTTP://"))    // extensões consideraremos que o
49        localização.startsWith("https://") || // protocolo é http
50        localização.startsWith("HTTPS://"))
51      return true;
52    else return false;
53    }
54
55 /**
56  * O método toString, que simplesmente retorna a string encapsulada por esta classe.
57  * @return a string encapsulada por esta classe
58  */
59  public String toString()
60    {
61    return localização;
62    }
63
64  } // fim da classe URL
```

A classe `DemoURL` (listagem 12.6) demonstra o uso dos métodos da classe `URL`.

Listagem 12.6: A classe `DemoURL`, que demonstra o uso de instâncias da classe `URL`.

```
1 /**
2  * A classe DemoURL, que demonstra o uso de instâncias da classe URL. Esta classe
3  * declara algumas instâncias e executa seus métodos.
4  */
```

```
5  class DemoURL // declaração da classe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém a declaração
9    * de algumas instâncias da classe URL, e as usa para executar métodos da classe.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     // Criamos algumas instâncias da classe URL.
16     URL yahoo = new URL("http://www.yahoo.com");
17     URL ftpCdrom = new URL("ftp://ftp.cdrom.com");
18     URL tulipas = new URL("http://www.bioinfo.univap.br/"+
19                           "~rafael/Images/Fotos/img0001.jpg");
20     // Usamos os métodos para analisar as URLs.
21     System.out.println(yahoo.éHTTP());      // true
22     System.out.println(yahoo.éImagem());    // false
23     System.out.println(ftpCdrom.éHTTP());   // false
24     System.out.println(ftpCdrom.éImagem()); // false
25     System.out.println(tulipas.éHTTP());    // true
26     System.out.println(tulipas.éImagem());  // true
27     } // fim do método main
28
29   } // fim da classe DemoURL
```

Além dos métodos para comparação de strings apresentados nesta seção, temos ainda dois métodos para comparação que não retornam simplesmente `true` ou `false`, mas retornam um valor correspondendo à ordem lexicográfica da string que chama o método em relação à passada como argumento.

*método* `compareTo`

O primeiro destes métodos é o método `compareTo`, que retorna um valor negativo caso a string que chamou o método esteja lexicograficamente antes da string passada como argumento, um valor positivo caso esteja depois e zero caso as duas strings sejam exatamente iguais.

A comparação é feita caracter a caracter, sendo que o valor retornado pelo método é a diferença entre os valores dos dois primeiros caracteres divergentes. Desta forma, se a string `estaPalavra` valer "`algoritmo`" e a string `outraPalavra` valer "`algarismo`" a expressão `estaPalavra.compareTo(outraPalavra)` retornará 14 (correspondente ao valor de `'o'-'a'`). Se os comprimentos das strings forem diferentes mas uma for prefixo da outra, o método retornará a diferença entre seus comprimentos: se `estaPalavra` valer "`algarismo`" e a string `outraPalavra` valer "`alga`" a expressão `estaPalavra.compareTo(outraPalavra)` retornará 5 (correspondente ao valor de `estaPalavra.length()-outraPalavra.length()`). Vale a pena notar que para quaisquer strings `a` e `b`, `a.compareTo(b)` será igual a `-b.compareTo(a)`.

*método* `compareTo-IgnoreCase`

O segundo método de comparação lexicográfica é o `compareToIgnoreCase`, que funciona da mesma maneira que o `compareTo`, mas considera caracteres maiúsculos e minúsculos como sendo iguais. Assim, se a string `umNome` for igual a "`Luís Humberto`" e a string `outroNome` for igual a "`luís Roberto`", o resultado da expressão `umNome.compareToIgnoreCase(outroNome)` será $-10$ (igual a `'H'-'R'`).

### 12.2.4 Métodos para procura em Strings

*método* `indexOf`

Frequentemente também é necessário verificar se uma string contém outra, ou em que posição uma string aparece dentro de outra. A classe `String` tem vários métodos para buscas de substrings (consideradas "trechos" de strings). O método de uso mais comum é o método `indexOf`, que recebe uma string como argumento e retorna a posição onde esta string aparece dentro da



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

string que executou o método, ou $-1$ se a string passada como argumento não estiver contida da que executou o método. Assim, se a string `dna` for igual a "`CTAGGTGTAGGC`" e a string `trecho` for igual a "`GG`", a expressão `dna.indexOf(trecho)` retornará 3 (a quarta posição, considerada a partir do zero, é onde a substring aparece na string). Vale a pena notar que a string `trecho` aparece mais de uma vez na string `dna`, mas somente a primeira ocorrência será considerada (o método possui outras formas que permitem verificar outras substrings além da primeira).

O método `indexOf` é frequentemente usado para verificar se uma substring é parte de outra: se o valor retornado for maior ou igual a zero, a substring está contida na string. A classe `EpocaDeFrutas`, na listagem 12.7, demonstra o uso do método `indexOf` para procura de substrings dentro de strings.

Listagem 12.7: A classe `EpocaDeFrutas`, que permite a consulta a um banco de dados primitivo sobre frutas e suas épocas.

```
1  /**
2   * A classe EpocaDeFrutas, que permite a consulta a um banco de dados primitivo sobre
3   * frutas e suas épocas.
4   */
5  class EpocaDeFrutas
6    {
7   /**
8    * Esta classe contém alguns campos estáticos que serão usados nas pesquisas.
9    */
10
11   // Array de strings contendo nomes de frutas
12   private static String[] frutas =
13     {"abacate","abacaxi","ameixa","banana","cereja","damasco","figo","laranja",
14      "limão","maçã","manga","melancia","morango","papaia","pêra","pêssego",
15      "tangerina","uva"};
16
17   // Este campo servirá como um atalho para o array seguinte, desta forma não será
18   // necessário declarar vários elementos do array com a mesma string.
19   private static String anoInteiro = "janeiro,fevereiro,março,abril,maio,junho,"+
20                                      "julho,agosto,setembro,outubro,novembro,dezembro";
21
22   // Array de meses, onde cada campo corresponde (inclusive na ordem) aos campos
23   // declarados no array de nomes de frutas.
24   private static String[] épocas =
25     {
26     anoInteiro,
27     anoInteiro,
28     "outubro,novembro,dezembro,janeiro",
29     anoInteiro,
30     "setembro,outubro,novembro,dezembro",
31     "setembro,outubro,novembro,dezembro,janeiro,fevereiro",
32     "dezembro,janeiro,fevereiro,março,abril",
33     anoInteiro,
34     "janeiro,fevereiro,março,abril,maio,junho,julho,agosto",
35     "dezembro,janeiro,fevereiro",
36     "outubro,novembro,dezembro,janeiro,fevereiro,março",
37     "outubro,novembro,dezembro,janeiro,fevereiro,março",
38     "abril,maio,junho,julho,agosto,setembro,outubro,novembro,dezembro",
39     anoInteiro,
40     "dezembro,janeiro,fevereiro,março,abril",
41     "setembro,outubro,dezembro,janeiro,fevereiro",
42     "abril,maio,junho,julho,agosto,setembro",
43     "novembro,dezembro,janeiro,fevereiro,março"
44     };
45
46   /**
47    * O método main permite a execução desta classe. Este método processa a linha de
48    * comando, considerando que somente um argumento será passado e que este argumento
49    * será um nome de mês. O método então pesquisa o argumento no array de meses. Onde
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
50   * houver correspondência, mostra a(s) fruta(s) que podem ser colhidas no mês
51   * correspondente ao argumento.
52   * @param argumentos uma lista de argumentos que serão os meses a serem pesquisados.
53   *        Este método considera que a lista conterá somente um elemento.
54   */
55  public static void main(String[] argumentos)
56    {
57    if (argumentos.length != 1) // nenhum argumento foi passado, imprimimos mensagem
58      {                         // e terminamos o programa
59      }
60    else // um argumento foi passado, será procurado no array de épocas
61      {
62      for(int qualÉpoca=0;qualÉpoca<épocas.length;qualÉpoca++)
63        {
64        // Procuramos a posição da String representada por argumentos[0] dentro da
65        // String representada pelo elemento épocas[qualÉpoca]. Se a posição for maior
66        // que zero, o mês aparece na lista representada pelo elemento do array épocas.
67        if (épocas[qualÉpoca].indexOf(argumentos[0]) >= 0)
68          {
69          // Imprimimos a fruta correspondente à época
70          System.out.println(frutas[qualÉpoca]);
71          }
72        } // fim do for para todos as épocas
73      }
74    } // fim do método main
75
76  } // fim da classe EpocaDeFrutas
```

Um ponto interessante da classe `EpocaDeFrutas` (listagem 12.7) é a forma como o método `indexOf` é usado: `épocas[qualÉpoca].indexOf(argumentos[0])` - como todos os elementos do array `épocas` são instâncias da classe `String`, podemos executar métodos desta classe em qualquer elemento do array (exceto os inicializados explicitamente com `null`).

Existe outra versão sobrecarregada do método `indexOf`, que aceita como argumento um único caracter ao invés de uma string, e retorna a posição de ocorrência deste primeiro caracter na string que executou o método, ou $-1$ se o caracter não ocorrer na string.

*método `indexOf` (com segundo argumento)*

Ainda outra versão do método `indexOf` aceita um segundo argumento do tipo `int` que indica que a busca deve ser feita a partir de certa posição. Assim, se a string `dna` for igual a `"CTAGGTGTAGGC"` e a string `trecho` for igual a `"GG"`, a expressão `dna.indexOf(trecho,0)` retornará 3 (localizou a substring na quarta posição), a expressão `dna.indexOf(trecho,4)` retornará 7 (localizou a substring na oitava posição, pois começou a procurar a partir da quinta), e a expressão `dna.indexOf(trecho,10)` retornará $-1$ (não localizou a substring, pois começou a procurar a partir da décima primeira posição). O segundo argumento para o método pode ser maior do que o comprimento da string ou até mesmo negativo, sem causar erros ou exceções. Existe também uma versão deste método que recebe como argumentos um caracter e um valor inteiro, com o mesmo comportamento.

*método `lastIndexOf`*

Um último método permite a procura em strings, mas efetuando a procura do final da string (último caracter) para o início. O método `lastIndexOf` recebe uma string como argumento e retorna a posição onde esta string foi encontrada na string que executou o método, ou $-1$ se não for encontrada, mas efetuando a busca do final da string para o início. Assim, se a string `proteína` for igual a `"LALGATPQILTPE"` e a string `trecho` for igual a `"TP"`, a expressão `proteína.lastIndexOf(trecho)` será igual a 10 (a última posição onde `trecho` aparece em `proteína`).

O método `lastIndexOf` também aceita um segundo argumento inteiro, correspondente à posição inicial de procura, de forma similar ao método `indexOf`. Assim, para as strings dadas



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

como exemplo anteriormente, a expressão `proteína.lastIndexOf(trecho,8)` retornará 5 e a expressão `proteína.lastIndexOf(trecho,2)` retornará $-1$ (não foram localizadas substrings antes do índice 2). Existem versões do método `lastIndexOf` que aceitam um caracter para procura, ao invés de uma string, de forma similar ao método `indexOf`.

### 12.2.5 Métodos para modificação de Strings

Instâncias da classe `String`, depois de criadas, são imutáveis - seu conteúdo não pode ser modificado diretamente (através de acesso aos caracteres individuais da string) ou indiretamente (através de métodos). Até mesmo operações de concatenação como `String nome = "Jean-Paul"; nome = nome + " Tremblay";` causam a criação de uma string temporária, contendo a concatenação das strings `nome` e `" Tremblay"`, que depois serão associadas à referência `nome`.

*método* `toLowerCase`

Alguns métodos da classe `String` permitem o retorno de uma versão modificada da string que os executar, de forma que a string que executar os métodos permanecerá inalterada. Dois destes métodos que são frequentemente usados, especialmente em conjunto com os métodos de comparação e busca de substrings, são os métodos `toLowerCase` e `toUpperCase`. O método `toLowerCase` não recebe argumentos e transforma todos os caracteres da string que o executar em minúsculos (inclusive respeitando acentos), retornando a string modificada. Desta forma, se a string `curso` valer `"Ciência da Computação"`, o resultado de `curso.toLowerCase()` será `"ciência da computação"`. Similarmente, o método transforma todos os caracteres da string que o executar em caracteres maiúsculos, retornando esta string transformada, de forma que o resultado da expressão `curso.toUpperCase()` será `"CIÊNCIA DA COMPUTAÇÃO"`.

*método* `toUpperCase`

*método* `trim`

Outro método que retorna uma versão modificada da string que o executar é o método `trim`, que elimina os espaços (e caracteres de controle) no início e fim de uma string, retornando a string sem estes caracteres. Por exemplo, se o campo `textoEntrado` valer `" Andrade\n\n"` (notem os dois espaços no início da string e as duas quebras de linhas no final), a expressão `textoEntrado.trim()` retornará `"Andrade"`, tendo eliminado quatro caracteres nas pontas da string original. O método `trim` não elimina espaços internos na string: se executássemos este método na string `"Macunaíma\nO herói sem nenhum caráter"` o resultado seria uma nova string idêntica à original, já que nenhum espaço ou caracter de controle foi removido.

*método* `replace`

Outro método que retorna uma string modificada a partir da string que executa o método é o método `replace`, que recebe como argumentos dois caracteres, e troca todas as ocorrências do primeiro caracter pelo segundo caracter, retornando a string modificada. Desta forma se a string `gíria` valesse `"pé-de-meia"`, o resultado da expressão `gíria.replace('-',' ')` seria `"pé de meia"` (todos os caracteres `'-'` foram trocados por espaços). Não é possível, com o método `replace`, eliminar ou inserir caracteres adicionais, uma vez que os dois argumentos para o método devam conter um único caracter.

*método* `substring`

Um método com duas formas permite a seleção e retorno de parte da string que o executar: o método `substring`, na forma que recebe somente um valor inteiro como argumento, retornará a substring composta por todos os caracteres cujos índices sejam maiores ou iguais ao valor passado. Assim, se a string `aminoácido` valer `"Fenilalanina"`, o resultado da expressão `aminoácido.substring(5)` será `"alanina"`.

O método `substring` também tem uma forma na qual dois valores numéricos são passados como argumentos, e o método retornará uma nova string composta de todos os caracteres cujos índices forem maiores ou iguais ao primeiro argumento e menores do que o segundo argumento. Desta forma, usando a string `aminoácido` como exemplo, o resultado da expressão

`aminoácido.substring(5,8)` será "`ala`". Os argumentos para o método devem estar entre zero e o comprimento da string menos um, e o segundo argumento deve ser maior que o primeiro, caso contrário uma exceção ocorrerá.

*método concat*

Um último método de modificação de strings digno de nota é o método `concat`, que recebe como argumento uma string e retorna a concatenação da string que executou o método com a passada como argumento. Este método tem a mesma funcionalidade do operador `+`, de forma que se a string `nome` valer "`Mário`", o resultado da operação `nome.concat(" de Andrade")` será "`Mário de Andrade`". A vantagem do uso do método `concat` sobre o operador `+` é que podemos fazer chamadas em cascata usando este método e outros da classe `String` caso seja necessário.

*chamada de métodos em cascata*

É interessante notar que como o resultado destes métodos é uma nova string, outros métodos da classe `String` podem ser chamados neste resultado diretamente, usando o ponto. Assim, podemos executar o método `curso.trim().toUpperCase()`, que eliminará os espaços nas pontas da string `curso`, logo em seguida convertendo os caracteres deste resultado para maiúsculas. Vale a pena notar que a ordem e o aninhamento dos métodos da classe `String` em uma chamada em cascata é importante: `"r".toUpperCase().concat("io de ").concat("j".toUpperCase()).concat("aneiro")` resultará em "`Rio de Janeiro`", enquanto que `"r".toUpperCase().concat("io de ").concat("j").toUpperCase().concat("aneiro")` resultará em "`RIO DE Janeiro`".

### 12.2.6 Métodos de conversão da classe `String`

*método valueOf*

A classe `String` contém ainda várias formas de um método de conversão de valores de tipos nativos para strings. Este método chama-se `valueOf` e é declarado como sendo estático, desta forma não precisamos criar instâncias da classe `String` para usá-lo. Este método converte valores de qualquer um dos tipos nativos, passados como argumentos, para uma string. O método também converte arrays de caracteres para strings, e instâncias de classes em geral para strings (através da chamada implícita ao método `toString` destas classes).

Ainda existem outros métodos da classe `String`, verifique a documentação do seu compilador Java e as APIs.

## 12.3 A classe `StringBuffer`

Conforme comentado, depois de criadas, instâncias da classe `String` não podem ser modificadas - os métodos da classe, como `toUpperCase` e `substring` criam sempre cópias modificadas da string original, retornando-as. Este processo pode ser computacionalmente custoso, em especial se modificações na string forem feitas repetidamente.

*classe StringBuffer*

Java fornece uma classe chamada `StringBuffer` que permite a criação e manipulação de uma string modificável. Se por um lado a criação de uma instância de `StringBuffer` é computacionalmente mais custosa do que a criação de uma instância de `String`, caso a string encapsulada por uma destas classes deva ser modificada frequentemente (em especial com a deleção e inserção de caracteres), isto será feito mais eficientemente pela instância da classe `StringBuffer`.

A classe `StringBuffer` é usada até mesmo pelo compilador JDK para otimizar trechos de código onde strings são concatenadas. Se no código sendo compilado houver o trecho `resultado = "o custo total é "+reais+" reais";` este trecho de código será compilado como sen-



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

do o equivalente ao trecho `resultado = new StringBuffer().append("o custo total é ").append(reais).append(" reais").toString();`.

Uma característica interessante da classe `StringBuffer` é a possibilidade de criação de instâncias com um número inicial fixo de caracteres, que pode ser modificado manualmente ou automaticamente, dependendo da necessidade de crescimento da string. Esta característica garante a eficiência da classe: memória adicional somente será alocada quando necessária, diferentemente do que é feito com a classe `String`, cujas modificações sempre exigem nova alocação de memória para novas strings.

*comprimento de instâncias de* `StringBuffer`

A classe `StringBuffer` tem então dois valores inteiros associados à ela, o primeiro sendo o *comprimento* que é o número de caracteres contido na instância, e que pode aumentar ou diminuir através da inserção, concatenação ou deleção de caracteres, valores de tipos nativos ou mesmo instâncias da classe `String` na string sendo encapsulada pela classe `StringBuffer`.

*capacidade de instâncias de* `StringBuffer`

O segundo valor associado a instâncias de `StringBuffer` é a *capacidade* que corresponde ao número máximo de caracteres que podem ser representados pela instância em um dado momento. A capacidade também pode ser modificada, explicitamente pelo programador ou automaticamente se tentarmos acrescentar caracteres ou valores de outros tipos de dados fazendo com que o comprimento atual exceda a capacidade atual.

### 12.3.1 Métodos básicos da classe `StringBuffer`

*construtores para a classe* `StringBuffer`

Novas instâncias da classe `StringBuffer` devem ser construídas através da palavra-chave `new` e da chamada de um dos construtores da classe. A classe tem três construtores sobrecarregados: um que não espera nenhum argumento e que constrói uma instância da classe `StringBuffer` sem caracteres (comprimento zero caracteres) mas com uma capacidade inicial de 16 caracteres. O segundo construtor espera como argumento uma instância da classe `String`, e cria a instância de `StringBuffer` usando os caracteres da string passada como argumento. Neste caso, o comprimento será o comprimento da string encapsulada e a capacidade será igual a este comprimento mais dezesseis caracteres. O terceiro construtor espera como argumento um valor inteiro correspondente à capacidade inicial da instância de `StringBuffer`, e o comprimento da string encapsulada será zero caracteres.

*método* `length`

O comprimento de uma string encapsulada em uma instância da classe `StringBuffer` pode ser recuperado através do método `length` (que não recebe argumentos, sendo equivalente ao método `length` da classe `String`), que retorna um número inteiro igual ao número de caracteres na string encapsulada. A capacidade de armazenamento da instância, em número de caracteres, também pode ser recuperada com o método `capacity`, que também retorna um valor inteiro.

*método* `capacity`

*método* `append`

Uma instância da classe `StringBuffer` pode ser modificada através de quatro mecanismos. O primeiro é a concatenação de valores no final do `StringBuffer`, funcionando de forma similar à do método `concat` da classe `String`. Qualquer tipo de dado pode ser concatenado ao final de uma instância de `StringBuffer`: existem métodos na classe para concatenação de valores dos tipos `boolean`, `char`, `int`, `long`, `float`, `double`, assim como arrays de caracteres (inteiros ou somente trechos) e instâncias da classe `String`, ou de qualquer classe que implemente o método `toString` (praticamente todas, já que todas as classes herdam da classe `Object` e esta tem o método `toString`). Quando um dado qualquer é concatenado a uma instância da classe `StringBuffer`, o seu comprimento é modificado. Se o comprimento exceder a capacidade atual do `StringBuffer`, a capacidade é aumentada para dezesseis mais o comprimento atual.

*método* `insert`

O segundo mecanismo de modificação de instâncias da classe `StringBuffer` é a inserção de dados em uma posição determinada da instância, e é feita pelo método `insert`, que recebe dois argumentos, o primeiro sendo a posição onde o dado deve ser inserido, e o segundo é o dado a ser inserido, que pode ser qualquer um dos tipos aceitos pelo método `append`. A posição de inserção deve ser um valor positivo, caso seja zero, o dado será inserido no início do `StringBuffer`. Os caracteres já existentes no `StringBuffer` não são sobreescritos, sendo descolados para o final para que a inserção não apague nada que já tenha sido colocado no `StringBuffer` anteriormente. Se o comprimento após a inserção exceder a capacidade atual do `StringBuffer`, a capacidade é aumentada para dezesseis mais o comprimento atual.

Os métodos `append` e `insert` são demonstrados pelos métodos da classe `StringUtils`, mostrada na listagem 12.8.

Listagem 12.8: A classe `StringUtils`, que contém alguns métodos que processam strings que não existem na classe `String`.

```
/**
 * A classe StringUtils, que contém alguns métodos que processam strings que não
 * existem na classe String. Estes métodos são implementados como métodos estáticos
 * e usam internamente métodos da classe StringBuffer para implementar os algoritmos.
 */
class StringUtils
  {

 /**
  * O método replace, que recebe como argumentos três strings, e troca todas as
  * ocorrências da segunda string pela terceira na primeira string. Este método
  * complementa o método replace na classe String, mas ao invés de substituir
  * caracteres por caracteres substitui strings por strings.
  * @param original a string original que sofrerá as substituições
  * @param de a string que será procurada para substituições
  * @param para a string que substituirá a string passada como argumento "de".
  * @return uma string onde todas as ocorrências de "de" foram trocadas por "para".
  */
  public static String replace(String original,String de,String para)
    {
    // Criamos uma instância de StringBuffer com capacidade inicial para o mesmo
    // número de caracteres da string original.
    StringBuffer resultado = new StringBuffer(original.length());
    // Para cada caracter na string original, vemos se ele é o início da string
    // "de". O laço não percorrerá todos os caracteres da string original, deixando
    // de lado os que não poderão ser iguais à string procurada por causa do tamanho
    // desta.
    for(int posição=0;posição<original.length()-de.length()+1;posição++)
      {
      // Comparamos a string que começa nesta posição com a "de".
      if (de.equals(original.substring(posição,posição+de.length())))
        {
        // Se são iguais, juntamos a string "para" ao StringBuffer.
        resultado.append(para);
        // Precisamos evitar que os outros caracteres que foram substituídos sejam
        // analisados pelo laço, pulando o seu processamento.
        posição = posição + de.length() - 1;
        }
      else // Se não, juntamos o caracter naquela posição.
        {
        resultado.append(original.charAt(posição));
        }
      }
    // Retornamos o resultado, convertido para uma instância da classe String.
    return resultado.toString();
    } // fim do método replace

 /**
```

```
49   * O método ordena, que recebe como argumento uma string e retorna uma string
50   * contendo os caracteres da primeira ordenados.
51   * @param original a string original que será ordenada
52   * @return uma string com os caracteres da string passada como argumento, ordenados.
53   */
54   public static String ordena(String original)
55     {
56     // Criamos uma instância de StringBuffer com capacidade inicial para o mesmo
57     // número de caracteres da string original.
58     StringBuffer resultado = new StringBuffer(original.length());
59     // Para o algoritmo iniciar adequadamente, usamos o primeiro caracter da string
60     // original para inicializar o StringBuffer.
61     resultado.append(original.charAt(0));
62     // Para cada caracter na string original (exceto o primeiro), vemos em que
63     // posição ele deve ser inserido na string resultante.
64     for(int posição=1;posição<original.length();posição++)
65       {
66       // Se o caracter naquela posição for menor ou igual ao primeiro caracter do
67       // StringBuffer, inserimos o caracter no início do StringBuffer.
68       if (original.charAt(posição) <= resultado.charAt(0))
69         resultado.insert(0,original.charAt(posição));
70       // Se o caracter naquela posição for maior ou igual ao último caracter do
71       // StringBuffer, inserimos o caracter no final do StringBuffer.
72       else if (original.charAt(posição) >= resultado.charAt(resultado.length()-1))
73         resultado.append(original.charAt(posição));
74       // Se não for um dos extremos, temos que procurar, no StringBuffer, em que
75       // posição este caracter deverá ser inserido. Basta localizar o primeiro
76       // caracter maior que este e inserí-lo antes.
77       else
78         {
79         for(int outraPosição=0;outraPosição<resultado.length();outraPosição++)
80           {
81           // Se achar um caracter no StringBuffer maior que o caracter sendo
82           // procurado,
83           if (resultado.charAt(outraPosição) > original.charAt(posição))
84             {
85             // Insere o caracter naquela posição e termina a busca.
86             resultado.insert(outraPosição,original.charAt(posição));
87             break;
88             }
89           } // terminou a procura da posição de inserção
90         }
91       } // terminou de analisar os caracteres da string original
92     // Retornamos o resultado, convertido para uma instância da classe String.
93     return resultado.toString();
94     } // fim do método ordena
95
96   } // fim da classe StringUtils
```

*método* `delete`

Um outro mecanismo de modificação de instâncias da classe `StringBuffer` é a deleção de caracteres ou trechos da instância. Isto pode ser feito usando os métodos `delete`, que aceita como argumentos dois números inteiros, e elimina todos os caracteres do `StringBuffer` que estejam entre o primeiro argumento e o último argumento menos um, reduzindo o comprimento da instância. Assim, a execução do trecho de código `StringBuffer palavra = new StringBuffer("antecipação"); System.out.println(palavra.delete(1,8));` causará a remoção dos caracteres com índices 1 a 7 e imprimirá "`ação`". Caso os dois argumentos sejam iguais, nada será feito. Caso o segundo argumento seja maior que o comprimento do `StringBuffer`, os caracteres do primeiro argumento até o último do `StringBuffer` serão eliminados, sem mensagem de erro. Caso qualquer um dos argumento seja negativo, a exceção `StringIndexOutOfBoundsException` será lançada. Caso o primeiro argumento seja maior que o segundo ou maior que o comprimento do `StringBuffer`, a mesma exceção ocorrerá.

Caracteres também podem ser deletados com o método deleteCharAt, que recebe como argumento um valor inteiro e elimina o caracter naquela posição no StringBuffer. Se o valor passado como argumento for negativo ou maior ou igual ao comprimento do StringBuffer, a exceção StringIndexOutOfBoundsException ocorrerá.

*método* deleteCharAt

O mecanismo final de modificação da string encapsulada em uma instância de StringBuffer é a alteração de caracteres, que pode ser feita através de dois métodos. O primeiro método, replace, recebe como argumentos dois valores inteiros correspondentes às posições inicial e final e uma instância da classe String. O método removerá todos os caracteres que estejam entre o primeiro argumento e o último argumento menos um, simulando uma chamada ao método delete, e inserirá a string passada como terceiro argumento na posição especificada pelo primeiro argumento. Por exemplo, o trecho de código `StringBuffer palavra = new StringBuffer("reação"); System.out.println(palavra.replace(2,3,"mo"));` imprimirá como resultado "remoção". O primeiro argumento não pode ser negativo ou maior ou igual ao comprimento do StringBuffer, nem o segundo argumento pode ser negativo, caso contrário a exceção StringIndexOutOfBoundsException ocorrerá.

*método* replace

Outro método de alteração de caracteres em um StringBuffer é o método setCharAt, que recebe como argumento uma posição (do tipo int) e um caracter, e modifica o caracter na posição especificada para ser gual ao caracter passado como argumento. O comprimento da instância de StringBuffer não é modificado. A posição não pode ser negativa ou maior ou igual ao comprimento do StringBuffer.

*método* setCharAt

A classe StringBuffer ainda contém alguns métodos interessantes. O método reverse coloca todos os caracteres do StringBuffer em ordem reversa, de forma que o primeiro passa a ser o último e vice-versa. A sequência original dos caracteres é perdida mas pode ser recuperada chamando de novo o método reverse na mesma referência. Existem também duas versões do método substring, que retornam trechos da string encapsulada como instâncias da classe String. A primeira versão do método recebe como argumento um valor inteiro, e retorna todos os caracteres cujos índices sejam iguais ou maiores a este valor, concatenados. A segunda versão recebe dois valores e retorna somente os caracteres cujos índices sejam maiores ou iguais ao primeiro valor e menores do que o segundo valor. Ambos valores devem ser positivos, e o primeiro sempre menor do que o comprimento da string encapsulada.

*método* reverse

*método* substring

A classe ainda tem dois métodos que permitem alterar seu tamanho e capacidade depois que a instância for criada. O método setLength modifica o comprimento da string encapsulada, recebendo como argumento um valor do tipo int. Se este valor for menor do que o comprimento atual da string encapsulada, esta será truncada até ficar com o comprimento especificado - os caracteres com índices maiores do que o valor passado simplesmente serão eliminados. Se o valor for maior do que o comprimento atual, o comprimento será modificado para o valor passado, e a string será preenchida com caracteres nulos (cujo valor é zero). A capacidade somente será modificada se o comprimento especificado for maior que a capacidade atual.

*método* setLength

O métodoensureCapacity permite a manipulação direta da capacidade da instância da classe StringBuffer. Este método recebe como argumento um valor inteiro e modifica a capacidade da instância usando este valor passado ou o dobro da capacidade atual mais dois, o que for maior. Por exemplo, se uma instância da classe StringBuffer for construída como `StringBuffer números = new StringBuffer("1");`, ela terá comprimento 1 e capacidade 17 (dezesseis a mais que o comprimento inicial). Se executarmos o trecho `números.ensureCapacity(30);`, a capacidade da instância será o maior de 30 e 17*2+2, ou 36. Se a capacidade especificada for

*método* ensureCapacity

negativa, nada será feito.

## 12.4 A classe `StringTokenizer`

Em rotinas de processamento de textos é frequente a necessidade de separar um texto em trechos que tem algum significado especial usando caracteres ou sequências que são consideradas separadoras. Um exemplo típico é a separação de uma string que representa uma data do tipo `"DD/MM/AAAA"` em seus componentes `DD`, `MM` e `AAAA`, correspondentes respectivamente ao dia, mês e ano.

tokens

Os componentes de interesse de uma string são conhecidos como *tokens*. A operação de quebra da string em *tokens* e separadores e subsequente análise é conhecida como *parsing* ou análise léxica. Podemos usar como exemplos de análise léxica a análise que um compilador faz de um código-fonte, para verificação de erros no código.

*classe* `StringTokenizer`

Java possui uma classe simples para quebra de strings em componentes e processamento individual destes componentes. A classe `StringTokenizer`, parte do pacote `java.util`, separa uma instância da classe `String` em uma série de *tokens* usando separadores *default* (espaços) ou especificados pelo programador. Os *tokens* podem ser recuperados, um a um, como instâncias da classe `String`, e convertidos para outros tipos de dados se necessário. Somente caracteres podem ser separadores: a classe `StringTokenizer` não separa *tokens* em strings usando outras strings como delimitadores.

*construtores para a classe* `StringTokenizer`

Instâncias da classe `StringTokenizer` podem ser construídas usando um dos três construtores. O primeiro construtor recebe como argumento somente a string a ser quebrada em *tokens*, e considera espaços, tabulações, quebras de linhas, retorno de carro e quebras de páginas (os caracteres `' '`, `'\t'`, `'\n'`, `'\r'` e `'\f'`), inclusive consecutivos como separadores (veja a tabela A.1 para mais informações). O segundo construtor recebe como argumentos uma string a ser quebrada em *tokens* e uma string contendo os caracteres que devem ser considerados como separadores. Cada um dos caracteres na segunda string será considerado um separador. O terceiro construtor recebe como argumentos duas strings, como o segundo construtor, só que recebe como um terceiro argumento um valor booleano que se for igual a `true`, considerará os separadores como também sendo tokens e que se for igual a `false` funcionará como o segundo construtor.

Um pequeno exemplo esclarecerá melhor as condições de quebra de strings quando cada cosnstrutor é usado. Se considerarmos a string `evento` como contendo `"Quinta-feira, 25 de outubro de 2001\nAniversário da Miyuki"`, e uma instância da classe `StringTokenizer` for criada usando o primeiro construtor e a string `evento` como argumento, esta instância conterá nove *tokens* (palavras separadas por espaços, quebras de linhas e semelhantes) e o primeiro *token* será `"Quinta-feira,"` (com a vírgula fazendo parte do *token*). Se uma segunda instância for criada usando o segundo construtor e como string de separadores a string `" ,"` (somente espaço e vírgula), a instância conterá oito tokens, sendo que o primeiro será `"Quinta-feira"` (sem a vírgula, que foi considerada separador), e o sexto será `"2001\nAniversário"`, já que o caracter `'\n'` não é mais considerado separador. Se uma outra instância for criada com o terceiro construtor, passando como primeiro argumento a string `evento`, como segundo argumento a string de separadores `" ,\n"` (espaço, vírgula e quebra de linha) e como terceiro argumento a constante booleana `true`, a instância conterá dezoito *tokens*, sendo que nove serão separadores.

*método* `countTokens`

Após a construção de uma instância da classe `StringTokenizer` a recuperação dos *tokens* poderá ser feita. Isto geralmente é feito com um laço, usando um contador e o resultado da execução

do método countTokens, que não recebe argumentos e retorna um valor inteiro igual ao número de *tokens* encontrado. A cada iteração deste laço, o método nextToken poderá ser usado. Este método recupera o próximo *token* como uma instância da classe String. É importante tomar cuidado com o uso dos métodos countTokens e nextToken - a cada *token* recuperado da instância da classe StringTokenizer o valor a ser retornado pelo método countTokens é decrementado, desta forma um laço for não deve usar uma comparação direta com countTokens para controlar o término do laço. A recuperação de *tokens* é feita um a um, sequencialmente, sem métodos que permitam recuperar o *token* anterior.

*método* nextToken

A iteração que permite a recuperação dos *tokens* pode ser também controlada usando o resultado da execução do método hasMoreTokens que não recebe argumentos e retorna o valor booleano true caso ainda existirem tokens a ser recuperados, e false caso contrário.

*método* hasMoreTokens

Como um exemplo de uso dos métodos da classe StringTokenizer, consideremos a classe DataHora, mostrada na listagem 12.9, cujo construtor recebe os dados como uma única string e quebra-a em seus componentes.

Listagem 12.9: A classe DataHora, cujo construtor recebe os campos como uma string formatada.

```
1  // É necessário importar este pacote para usar a classe StringTokenizer
2  import java.util.*;
3
4  /**
5   * A classe DataHora, que reusa as classes Data e Hora através de delegação.
6   * O construtor desta classe usa uma instância da classe StringTokenizer para
7   * obter os valores para os campos que estão formatados em uma string.
8   */
9
10 class DataHora // declaração da classe
11   {
12   /**
13    * Declaração dos campos da classe.
14    */
15   private Data estaData; // uma instância da classe Data representa o dia, mês e ano
16   private Hora estaHora; // uma instância da classe Hora representa a hora, minuto
17                          // e segundo
18
19  /**
20   * O construtor para a classe DataHora, que inicializa os campos da classe.
21   * Os campos são passados como formatados em uma string com o formato
22   * "dia/mês/ano hora:minuto:segundo".
23   * @param formatada a string formatada com os dados
24   */
25   DataHora(String formatada)
26     {
27     // Declaração de variáveis locais
28     byte aHora,oMinuto,oSegundo,oDia,oMês;
29     short oAno;
30     // A string formatada tem como separadores espaços, barra e dois-pontos.
31     // Criamos uma instância de StringTokenizer para quebrar a string usando estes
32     // separadores.
33     StringTokenizer campos = new StringTokenizer(formatada," :/");
34     // Esperamos seis tokens - caso não existam seis, vamos considerar
35     // a data como incorreta e preencher os campos com valores padrão.
36     if (campos.countTokens() != 6)
37       {
38       oDia = 1; oMês = 1; oAno = 1970;       // meia-noite de 1/1/1970
39       aHora = 0; oMinuto = 0; oSegundo = 0;
40       }
41     // Temos seis tokens, vamos pegar um por um e converter para tipos nativos.
42     else
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
43        {
44        oDia     = Byte.parseByte(campos.nextToken());
45        oMês     = Byte.parseByte(campos.nextToken());
46        oAno     = Short.parseShort(campos.nextToken());
47        aHora    = Byte.parseByte(campos.nextToken());
48        oMinuto  = Byte.parseByte(campos.nextToken());
49        oSegundo = Byte.parseByte(campos.nextToken());
50        }
51      // Com os campos preenchidos, inicializamos as instâncias.
52      estaData = new Data(oDia,oMês,oAno);
53      estaHora = new Hora(aHora,oMinuto,oSegundo);
54      }
55
56    /**
57     * O método toString não recebe argumentos, e retorna uma String contendo os valores
58     * dos campos da classe formatados. Os valores são obtidos através da chamada
59     * implícita aos métodos toString das instâncias das classes embutidas. Note que o
60     * formato retornado é o mesmo esperado pelo construtor da classe.
61     * @return uma String com os valores dos campos formatados.
62     */
63    public String toString()
64      {
65      return estaData+" "+estaHora;
66      }
67
68    } // fim da classe DataHora
```

Um outro exemplo mais complexo do uso de uma instância da classe `StringTokenizer` é mostrado na classe `CasamentoDePadroes`, mostrada na listagem 12.10. Esta classe encapsula um mecanismo simples de casamento de padrões, uma tarefa bastante comum em processamento de textos, que é descrita de forma simplificada a seguir.

Imagine que queiramos verificar a similaridade entre uma string e outra, mas com certas regras de relaxamento (por exemplo, considerando que as strings sejam iguais mesmo se certos caracteres em certas posições forem diferentes) ou substituição (considerando uma string como sendo igual a uma outra string que é um elemento de um conjunto de strings bem semelhantes). As aplicações de funções de comparação mais lenientes são muitas: reconhecimento de entrada de usuários mesmo com pequenas diferenças do esperado, localização de informação semelhante mas não exatamente igual à procurada, economia na descrição de strings para comparação, ferramentas de análise de texto, etc.

Para implementar funções de comparação inexata consideramos que ao invés de compararmos duas strings (ou uma string com um conjunto de outras) compararemos uma string com um *padrão*. Este padrão, também representado por uma string, contém informações adicionais que indicam que trechos da string poderão ser considerados iguais e sob que condições. Para nosso exemplo simplificado, consideraremos que trechos do padrão entre chaves (caracteres `'{'` e `'}'`) serão considerados listas de alternativas, que devem estar separadas entre si pelo caracter `'|'`.

Dizemos que uma string *casa* com o padrão se ela for igual a uma das combinações que podem ser feitas com as listas de alternativas do padrão. Desta forma, se considerarmos o padrão "`b{iologica|asica|ienal}mente`", podemos dizer que as strings "`biologicamente`", "`basicamente`" e "`bienalmente`" casam com o padrão, enquanto "`bravamente`" e "`aumente`" não casam com o padrão. O padrão pode ser descrito como representando qualquer palavra que começe com "b", termine com "mente" e tenha "iologica", "asica" ou "ienal" no meio.

O algoritmo que verifica se uma string casa com um padrão é relativamente simples, e descrito a seguir. Como entrada do algoritmo consideramos uma string e um padrão. Os passos são:

1. Quebramos o padrão em *tokens* usando os caracteres `'{'` e `'}'` como separadores, descartando os separadores. Cada *token* obtido pode ser uma lista de alternativas separadas pelo caracter `'|'` ou um elemento único (também conhecido como *singleton*).
2. Enquanto houverem *tokens* obtidos do padrão, repetimos os seguintes passos:
3. Se o token for um *singleton*, comparamos seus caracteres com os caracteres correspondentes da string. Se houver diferença, o casamento de padrões falhou. Se não houver, removemos os *N* primeiros caracteres da string de entrada, onde *N* é o comprimento do *singleton*, e continuamos (voltando para o passo 2).
4. Se o token for uma lista de alternativas, quebramos esta lista em *tokens* usando o caracter `'|'` como separador, descartando o separador. Inicializamos uma variável booleana `nenhumCasa` que representa o casamento da string com um elemento qualquer desta lista de alternativas com `true`.
5. Enquanto houverem *tokens* obtidos da lista de alternativas, repetimos os seguintes passos:
6. Comparamos os caracteres do *token* obtido da lista de alternativas com os caracteres correspondentes da string. Se houver igualdade, removemos os *N* primeiros caracteres da string de entrada, onde *N* é o comprimento do *token* obtido da lista de alternativas, modificamos a variável booleana `nenhumCasa` para `true` e interrompemos o processamento dos tokens da lista de alternativas.
7. Se ao final dos passos 5 e 6 a variável `nenhumCasa` ainda estiver valendo `true`, consideramos que o casamento de padrões falhou. Se a variável `nenhumCasa` valer `false`, continuamos do passo 2.
8. Se ao final do processamento dos *tokens* ainda houverem caracteres na string sendo comparada, consideramos que o casamento de padrões falhou.
9. Se o casamento de padrões tiver falhado, retornamos `false`, senão retornamos `true`.

A classe `CasamentoDePadroes`, que implementa este algoritmo, é mostrada na listagem 12.10.

Listagem 12.10: A classe `CasamentoDePadroes`, que implementa um mecanismo simples de casamento de padrões.

```
1  // É necessário importar este pacote para usar a classe StringTokenizer
2  import java.util.*;
3
4  /**
5   * A classe CasamentoDePadroes, que implementa um mecanismo simples de casamento de
6   * padrões. Esta classe encapsula um padrão simples e contém um método que verifica
7   * se uma dada string corresponde ao padrão.
8   */
9  class CasamentoDePadroes // declaração da classe
10   {
11   /**
12    * O campo da classe, uma única string que armazena o padrão.
13    */
14    private String padrão;
15
16   /**
17    * O construtor da classe, que recebe como argumento uma string e usa este
18    * argumento para inicializar o campo da classe.
19    * @param padrão o padrão a ser armazenado
20    */
21    CasamentoDePadroes(String padrão)
22      {
23      this.padrão = padrão;
24      }
25
26   /**
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
27    * O método casa, que verifica se o padrão casa com uma string passada como
28    * argumento. O algoritmo de casamento é explicado no texto.
29    * @param aComparar a string a ser comparada com o padrão.
30    * @return true se a string casa com o padrão, false caso contrário.
31    */
32    public boolean casa(String aComparar)
33      {
34      // Inicialmente consideramos que todos os tokens casam com os padrões.
35      boolean todosCasam = true;
36      // Estas strings armazenarão os tokens obtidos na string padrão e na que conterá
37      // uma lista de alternativas.
38      String esteToken,esteTokenDoGrupo;
39      // Criamos uma instância de StringTokenizer usando os delimitadores '{' e '}'.
40      StringTokenizer oPadrão = new StringTokenizer(padrão,"{}");
41      // Enquanto houverem tokens na string padrão...
42      while(oPadrão.hasMoreTokens())
43        {
44        esteToken = oPadrão.nextToken(); // recuperamos o token
45        // Se este token for um singleton, comparamos a string com o token,
46        // considerando somente o tamanho do token. Para ver se o token é um
47        // singleton, verificamos se ele contém ou não o caracter |.
48        if (esteToken.indexOf('|') == -1) // é um singleton !
49          {
50          // Se o token é igual ao início da string
51          if (aComparar.startsWith(esteToken))
52            {
53            // Reduzimos a string, eliminando o seu prefixo (=token)
54            aComparar = aComparar.substring(esteToken.length());
55            }
56          else // se não começa, o casamento de padrões falhou
57            todosCasam = false;
58          } // o token não é um singleton, mas uma lista de alternativas
59        else
60          {
61          // Criamos uma isntância de StringTokenizer para analisar a lista de
62          // alternativas, que devem estar separados pelo separador '|'.
63          StringTokenizer grupo = new StringTokenizer(esteToken,"|");
64          boolean nenhumCasa = true; // em princípio nenhum casa
65          // Enquanto houverem tokens na lista de alternativas...
66          while(grupo.hasMoreTokens())
67            {
68            esteTokenDoGrupo = grupo.nextToken(); // recuperamos o token
69            // Se o token é igual ao início da string
70            if (aComparar.startsWith(esteTokenDoGrupo))
71                {
72              // Reduzimos a string, eliminando o seu prefixo (=token)
73              aComparar = aComparar.substring(esteTokenDoGrupo.length());
74              nenhumCasa = false; // ao menos um deles casa com a string
75              break; // não precisamos verificar os outros
76              }
77            } // terminamos de ver os tokens da lista de alternativas
78          if (nenhumCasa) // se nenhum dos tokens da lista de alternativas casou...
79            todosCasam = false; // o casamento de padrões falhou
80          } // fim do processamento para o grupo de padrões
81        } // fim do while para todos os tokens cujos separadores foram '{' e '}'
82      // Se ao final dos tokens ainda restam caracteres na string a comparar, então
83      // consideramos que ela não casa com o padrão
84      if (aComparar.length() > 0)
85        todosCasam = false;
86      return todosCasam;
87      } // fim do método casa
88
89  /**
90   * O método toString, que simplesmente retorna o padrão encapsulado.
91   * @return o padrão encapsulada por esta classe
92   */
93   public String toString()
94     {
95     return padrão;
```

```
96      }
97
98    } // fim da classe CasamentoDePadroes
```

Alguns exemplos de uso de instâncias da classe `CasamentoDePadroes` são mostrados na classe `DemoCasamentoDePadroes`, na listagem 12.11.

Listagem 12.11: A classe `DemoCasamentoDePadroes`, que mostra exemplos de uso da classe `CasamentoDePadroes`.

```
1  /**
2   * A classe DemoCasamentoDePadroes, que mostra exemplos de uso da classe
3   * CasamentoDePadroes.
4   */
5  class DemoCasamentoDePadroes // declaração da classe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém a declaração
9    * de duas instâncias da classe CasamentoDePadroes, e as usa para verificar se
10   * algumas strings casam com aqueles padrões.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] args)
15     {
16     CasamentoDePadroes datas =
17       new CasamentoDePadroes("1{1|2|3|4|5} de {janeiro|fevereiro|março} de 200{1|2}");
18     System.out.println(datas.casa("15 de fevereiro de 2001")); // true
19     System.out.println(datas.casa("11 de março de 2002"));     // true
20     System.out.println(datas.casa("31 de janeiro de 2003"));   // false
21     CasamentoDePadroes horas = new CasamentoDePadroes("{00|04|08|12|16|20}:00:00");
22     System.out.println(horas.casa("12:00:00"));          // true
23     System.out.println(horas.casa("12:00:01"));          // false
24     System.out.println(horas.casa("12:00"));             // false
25     System.out.println(horas.casa("12:00:00 de hoje")); // false
26     System.out.println(horas.casa("às 12:00:00"));       // false
27     } // fim do método main
28
29   } // fim da classe DemoCasamentoDePadroes
```

## 12.5 Exercícios do capítulo 12

Os exercícios deste capítulo que envolvem conceitos de biologia computacional são, em geral, versões simplificadas de algoritmos e problemas reais.

Alguns dos algoritmos descritos nesta seção (e suas variações) podem ser conhecidos por outros nomes.

**Exercício 12.1:** ⋆

Após o término da execução do trecho de código `String s = "alfabeto"; char[] array = s.toCharArray();`, quais opções serão verdadeiras ?

**A.** `array[8]` será igual a zero.
**B.** `array[8]` será igual a `null`.
**C.** `s` será igual a `null`.
**D.** O comprimento de `array` será oito.
**E.** `s` permanecerá inalterada.

**Exercício 12.2:** ⋆

Quais dos seguintes trechos de código podem ser compilados sem erros ?

**A.** `char[] array; String texto = new String(array);`
**B.** `char[] array = new char[20]; String texto = new String(array);`
**C.** `int[] array = new int[20]; String texto = new String(array);`
**D.** `char[][] array = new char[10][10]; String texto = new String(array);`
**E.** `String texto = new String(Math.PI);`
**F.** `String texto = new String(""+Math.PI);`

**Exercício 12.3:** ⋆

Quais serão os valores retornados pelo método `length` quando aplicado às strings "`cadeia de caracteres`", "`\n\n`" e "" ?

**A.** 20, 2 e zero, respectivamente.
**B.** 19, 2 e um, respectivamente.
**C.** 18, 2 e zero, respectivamente.
**D.** 20, 4 e indefinido, respectivamente.
**E.** 19, 4 e `null`, respectivamente.
**F.** 18, 4 e zero, respectivamente.

**Exercício 12.4:** ⋆

Quais serão os valores retornados pelo método `length` quando aplicado ao resultado da execução do método `trim` nas strings "`cadeia de caracteres`", "`\n\n`" e "" ?

**A.** 20, 2 e `null`, respectivamente.
**B.** 18, 2 e -1, respectivamente.
**C.** 16, 2 e zero, respectivamente.
**D.** 20, zero e zero, respectivamente.
**E.** 18, zero e `null`, respectivamente.
**F.** 16, zero e zero, respectivamente.

**Exercício 12.5:** ⋆

Após a execução da linha `String[] seteAnões = {"Atchim", "Dengoso", "Dunga", "Feliz", "Mestre", "Soneca", "Zangado"};` em um método qualquer, quais das opções abaixo serão verdadeiras ?

**A.** O resultado da chamada do método `seteAnões[3].charAt(2)` será o caracter `'l'`.
**B.** O resultado da chamada do método `seteAnões[0].charAt(0)` será o caracter espaço.
**C.** O resultado da chamada do método `seteAnões[0].charAt(0)` causará uma erro de tempo de execução.
**D.** O resultado da chamada do método `seteAnões[2].charAt(5)` causará uma erro de tempo de execução.
**E.** O oitavo elemento do array é igual a `null`.

**Exercício 12.6:** ⋆

Quais das seguintes operações com strings abaixo retornará o valor booleano `true` ?

**A.** `"ABC".concat("DEF").equals("ABCDEF")`
**B.** `"ABC"+"DEF".equals("ABCDEF")`
**C.** `"ABC ".concat("DEF").trim().equals("ABCDEF")`
**D.** `"ABC ".trim().concat("DEF").equals("ABCDEF")`
**E.** `"abc".toUpperCase().concat("DEF").toLowerCase().equals.("ABCdef")`
**F.** `"abc".toUpperCase().concat("DEF".toLowerCase()).equals.("ABCdef")`

**Exercício 12.7:** ⋆

Quais das seguintes operações com strings abaixo retornará o valor booleano `true` ?

**A.** `"Tremblay".startsWith("T")`
**B.** `"Tremblay".endsWith("Y")`
**C.** `"Tremblay".toLowerCase().startsWith("tre")`
**D.** `"Tremblay".startsWith("tre".toUpperCase())`
**E.** `"Tremblay".trim().startsWith("re")`

**Exercício 12.8:** ⋆

Considerando a string `"alternativamente"` quais valores serão retornados pelas operações `indexOf("te")`, `indexOf("te",3)` e `indexOf("te",15)` ?

**A.** 2, 3 e 15, respectivamente.
**B.** 2, 14 e 15, respectivamente.
**C.** 2, 14 e -1, respectivamente.
**D.** 3, 14 e -1, respectivamente.
**E.** 3, 15 e -1, respectivamente.
**F.** 3, 3 e 15, respectivamente.

**Exercício 12.9:** ⋆

Considerando a string `"alternativamente"` quais valores serão retornados pelas operações `lastIndexOf("te")`, `lastIndexOf("te",3)` e `lastIndexOf("te",15)` ?

**A.** 14, 2 e -1, respectivamente.
**B.** 14, 2 e 3, respectivamente.
**C.** -1, 2 e -1, respectivamente.
**D.** 14, 3 e 3, respectivamente.
**E.** -1, 3 e 3, respectivamente.
**F.** 14, 2 e 14, respectivamente.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.10:** ⋆

Considerando a string `palavra` valendo "`rearrumasse`" qual será o resultado da expressão `palavra.replace('r','X').replace('a','Y').replace('S','Z')` ?

**A.** "`XeYXXumYZZe`"
**B.** "`rearrumasse`"
**C.** "`XeaXXumasse`"
**D.** "`rearrumaZZe`"
**E.** "`XeYXXumYsse`"

**Exercício 12.11:** ⋆

Considerando a string `palavra` valendo "`autodeterminação`" quais serão os resultados das expressões `palavra.substring(11)`, `palavra.substring(6,13)` e `palavra.substring(4,9)` ?

**A.** "`ação`", "`ermina`" e "`eter`", respectivamente.
**B.** "`inação`", "`erminaç`" e "`eterm`", respectivamente.
**C.** "`nação`", "`termina`" e "`deter`", respectivamente.
**D.** "`inação`", "`erminaç`" e "`eterm`", respectivamente.
**E.** "`nação`", "`ermina`" e "`eter`", respectivamente.
**F.** "`ação`", "`termina`" e "`deter`", respectivamente.

**Exercício 12.12:** ⋆

Considerando a string "`inconstitucionalidade e interdisciplinariedade`" e o trecho de código `StringTokenizer st1,st2,st3; st1 = new StringTokenizer(a,"i"); st2 = new StringTokenizer(a,"in"); st3 = new StringTokenizer(a," ",true);`, quantos tokens serão criados para as três instâncias da classe `StringTokenizer` ?

**A.** 11, 11 e 5, respectivamente.
**B.** 9, 11 e 5, respectivamente.
**C.** 11, 11 e 3, respectivamente.
**D.** 9, 3 e 3, respectivamente.
**E.** 11, 3 e 3, respectivamente.

**Exercício 12.13:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoString
2    {
3    public static void main(String[] argumentos)
4      {
5      String nome = "Dawkings";
6      nome.length = 4;
7      System.out.println(nome);
8      }
9    }
```

**Exercício 12.14:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoString
2    {
3    public static void main(String[] argumentos)
4      {
5      String nome = "Dan Gusfield";
6      nome.charAt(3) = '+';
7      System.out.println(nome);
8      }
9    }
```

**Exercício 12.15:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoString
2    {
3    public static void main(String[] argumentos)
4      {
5      String nome = "Jacques Bertin";
6      System.out.println(nome.charAt(-1));
7      System.out.println(nome.charAt(13));
8      System.out.println(nome.charAt(14));
9      }
10   }
```

**Exercício 12.16:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoString
2    {
3    public static void main(String[] argumentos)
4      {
5      StringBuffer mutável = "Podemos incluir caracteres em um StringBuffer ";
6      mutável += " de forma mais eficiente do que se usássemos uma String";
7      }
8    }
```

**Exercício 12.17:** ⋆

Identifique e explique o(s) erro(s) na classe abaixo.

```
1  class DemoString
2    {
3    public static void main(String[] argumentos)
4      {
5      URL univap = "http://www.univap.br/";
6      System.out.println(univap.éHTTP());
7      }
8    }
```

**Exercício 12.18:** ⋆

Modifique a classe `JogoDaForca` (listagem 12.1) para que o construtor filtre a string recebida, garantindo que a palavra a ser adivinhada não terá caracteres maiúsculos (ou seja, convertendo todos os caracteres da string para minúsculos).

**Exercício 12.19:** ⋆

Escreva, para a classe `URL` (listagem 12.5), um método `éHTML` que retorne `true` caso a URL termine com `htm` ou `html`.

**Exercício 12.20:** ⋆

Escreva, para a classe `URL` (listagem 12.5), um método `éFilme` que retorne `true` caso a URL termine com `mov`, `avi`, `rm` ou `mpeg`.

**Exercício 12.21:** ⋆⋆

Crie, na classe `StringUtils` (listagem 12.8), um método estático `desacentua` que recebe como argumento uma string, e que substitua todos os caracteres acentuados desta string por caracteres não-acentuados correspondentes. Por exemplo, se a string "`Nação`" for passada como argumento, este método deverá retornar "`Nacao`". O método deve considerar maiúsculas e minúsculas como sendo diferentes. *Dica:* várias chamadas ao método `replace`, em cascata, poderão resolver o problema.

**Exercício 12.22:** ⋆⋆

Crie, na classe `StringUtils`, o método `replica`, que recebe como argumentos uma string e um valor inteiro, e retorna uma string composta de várias repetições da string passada como argumento, onde o número de repetições deve ser o número passado como argumento. Por exemplo, se os argumentos para este método forem a string "`Ha!`" e o valor 3, o método deverá retornar "`Ha!Ha!Ha!`".

**Exercício 12.23:** ⋆⋆

Crie, na classe `StringUtils`, o método `alinhaÀDireita`, que recebe como argumentos uma string e um valor numérico, e completa a string com espaços à esquerda até que o comprimento da string fique igual ao valor numérico passado, retornando a string modificada. Escreva também o método `alinhaÀEsquerda`, que faz o mesmo mas adicionando espaços à direita. Se o comprimento da string passada já for maior que o valor passado como argumento, o método deve retornar a string inalterada.

**Exercício 12.24:** ⋆⋆

Escreva o método estático `alinhaAoCentro` na classe `StringUtils` que receba uma string e um valor inteiro como argumentos e adicione espaços no início e fim da string de forma que o comprimento final desta string seja igual ao valor passado como argumento. A string resultante deverá ser retornada. Por exemplo, se a String "`relatório`" e o valor 20 forem passados como argumentos, o método deverá adicionar espaços no início e no fim da string e retornar "  `relatório`  ". Se o comprimento da string passada já for maior que o valor passado como argumento, o método deve retornar a string inalterada.

**Exercício 12.25:** ⋆⋆

Escreva na classe `StringUtils` (listagem 12.8) um método estático `conta` que receba como argumentos uma string e um caracter, e retorne um inteiro correspondente ao número de ocorrências do caracter na string passados como argumentos.

**Exercício 12.26:** ⋆⋆

Escreva um programa em Java que receba uma string pelo teclado e imprima todas as ocorrências de duas letras iguais uma após a outra. Por exemplo, se este programa receber a string `"irrepreensível"`, deverá imprimir como resultado `"rr"` e `"ee"`. *Dica:* use um laço e o método `charAt`.

**Exercício 12.27:** ⋆⋆

Modifique a classe `JogoDaForca` (listagem 12.1) para que o construtor filtre a string recebida, garantindo que a palavra a ser adivinhada não terá acentos. Use, para isto, o método `desacentua` da classe `StringUtils` (veja exercício 12.21).

**Exercício 12.28:** ⋆⋆

O método `pergunta` da classe `JogoDaForca` (listagem 12.1) não verifica se o usuário entrou uma letra das que já foram tentadas anteriormente. Modifique este método para que a pergunta seja feita enquanto o usuário não entrar uma letra que não tenha sido utilizada anteriormente.

**Exercício 12.29:** ⋆⋆

Escreva uma classe em Java que represente o nome completo de uma pessoa, composto de três strings (nome próprio, nome do meio e nome da família). Escreva nessa classe o método `rubrica` que retorna somente as iniciais do nome completo em caracteres minúsculos, e o método `assinatura` que retorna as iniciais dos nomes próprio e do meio (com pontos) e o nome de família completo. Por exemplo, se o nome da pessoa representado por esta classe for `"Richard Langton Gregory"`, o método `rubrica` deve retornar `"rlg"` e o método `assinatura` deve retornar `"R.L.Gregory"`. Para facilitar, considere armazenar os três nomes em strings separadas.

**Exercício 12.30:** ⋆⋆

Modifique a classe `DemoLogin` (listagem 12.4) para que o usuário tenha três chances para digitar a senha encapsulada pela classe. Caso a senha não esteja correta após três tentativas, o programa deve imprimir uma mensagem e terminar a sua execução.

**Exercício 12.31:** ⋆⋆

Escreva na classe `StringUtils` vários métodos sobrecarregados (veja seção 4.3) para comparação entre strings, baseados nos métodos `equals` e `equalsIgnoreCase`, que recebam duas ou mais strings e retornem `true` se todas forem iguais e `false` se qualquer uma for diferente das outras. *Dica:* estes métodos podem ser chamados em cascata, de forma que o método que compara três strings pode chamar o método que compara duas e assim em diante.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.32:** ⋆⋆

Escreva na classe `StringUtils` (veja exercício 12.21) um método que reverta a ordem dos caracteres de uma string passada como argumento e retorne a string revertida. Um exemplo: se a string `"Java"` for passada para este método, ele deve retornar a string `"avaJ"`. *Dica:* Use um laço `for` ou `while` e o método `charAt`, e crie uma nova string que receberá os caracteres na ordem invertida. Não use mecanismos da classe `StringBuffer`.

**Exercício 12.33:** ⋆⋆

Implemente você mesmo um método com o mesmo funcionamento do método `toCharArray` na classe `StringUtils`.

**Exercício 12.34:** ⋆⋆

Modifique a classe `Login` (listagens 12.3) para ignorar espaços adicionais entrados pelo usuário no nome e senha, somente se a verificação não for rigorosa. Se a verificação não for rigorosa, a senha esperada for `"finlândia"` e o usuário entrar `" finlândia "`, o método `verifica` deverá retornar `true`.

**Exercício 12.35:** ⋆⋆

Modifique a classe `Login` (listagens 12.3) para que o processo de verificação considere também caracteres acentuados como sendo iguais aos não-acentuados. Por exemplo, se a verificação não for rigorosa, a senha esperada for `"lapônia"` e o usuário entrar `"laponia"`, o método `verifica` deverá retornar `true`. Veja também o exercício 12.21.

**Exercício 12.36:** ⋆⋆

Modifique a classe `URL` (listagem 12.5) para que todas as combinações possíveis de maiúsculas e minúsculas sejam consideradas nos métodos `éImagem` e `éHTTP`. *Dica:* mantenha uma versão da URL encapsulada somente com caracteres minúsculos, eliminando a necessidade de comparação com maiúsculos.

**Exercício 12.37:** ⋆⋆

Escreva, para a classe `URL` (listagem 12.5), um método `éScript` que retorne `true` se a localização terminar em `"cgi"`, `"asp"` ou `"php"`. Veja também o exercício 12.36.

**Exercício 12.38:** ⋆⋆

O método `replace` da classe `String` somente substitui um único caracter por vez. Escreva um método estático `replaceMultiple` na classe `StringUtils` (listagem 12.8) que receba três strings como argumentos: a primeira é a string que servirá de base para as modificações, a segunda string conterá os caracteres que serão substituídos pelos caracteres correspondentes na terceira string. O método deverá retornar uma string modificada usando os dados passados como argumentos. A segunda e terceira strings devem ter exatamente o mesmo número de caracteres, caso contrário o método deve retornar `null`. Por exemplo, se o método for chamado como `replaceMultiple("Paralelepípedo","alp","xyz")` o resultado deverá ser `"Pxrxyeyezízedo"` (`'a'` foram trocados por `'x'`, `'l'` foram trocados por `'y'` e `'p'` foram trocados por `'z'`).

**Exercício 12.39:** ⋆⋆

Escreva um programa em Java que compare a performance de várias chamadas ao método `concat` em cascata com várias chamadas ao operador `+`, por exemplo, executando os seguintes trechos de código:

- `String hexa = "a".concat("b").concat("c").concat("d").concat("e").concat("f");`
- `String a = "a"; String b = "b"; String c = "c"; String d = "d"; String e = "e"; String f = "f"; String hexa = a+b+c+d+e+f;`
- `String hexa = "a"+"b"+"c"+"d"+"e"+"f";`

Execute cada um destes trechos um milhão de vezes, para uma medida melhor de tempo gasto. Meça os resultados com o método `currentTimeMillis` da classe `System` (veja a seção 7.3). Explique a diferença de tempo entre as várias chamadas, se houver.

**Exercício 12.40:** ⋆⋆

Escreva um método estático `removeEspaçosInternos` na classe `StringUtils` (listagem 12.8) que, recebendo uma string como argumento, remova espaços múltiplos na string e retorne a string resultante. Por exemplo, se a string `"Uma string com muitos espaços redundantes"` for passada, o método deve retornar a string `"Uma string com muitos espaços redundantes"`.

**Exercício 12.41:** ⋆⋆

Em um jogo de tabuleiro chamado *Palavras Cruzadas*, cada palavra formada por um jogador vale um certo número de pontos, que depende das letras usadas. O número de pontos para as letras do alfabeto é dado por:

- Para cada letra `'Q'` ou `'Z'` na palavra, some 10 pontos.
- Para cada letra `'J'` ou `'X'` na palavra, some 8 pontos.
- Para cada letra `'K'` na palavra, some 5 pontos.
- Para cada letra `'F'`, `'H'`, `'V'`, `'W'` ou `'Y'` na palavra, some 4 pontos.
- Para cada letra `'B'`, `'C'`, `'M'` ou `'P'` na palavra, some 3 pontos.
- Para cada letra `'D'` ou `'G'` na palavra, some 2 pontos.
- Para todas as outras letras, some 1 ponto.

Por exemplo, o número de pontos para a palavra `"Java"` no jogo será $8+1+4+1=14$ pontos. Escreva uma classe `PalavrasCruzadas` em Java que contenha um método que receba uma string como argumento e retorne o número de pontos que esta string valeria, no jogo.

**Exercício 12.42:** ⋆⋆

O método `replace` da classe `String` considera caracteres maiúsculos e minúsculos como sendo diferentes: se a string `mágica` valer `"Abracadabra"` o resultado de `mágica.replace('a','o')` será `"Abrocodobro"` e não `"Obrocodobro"`. Escreva um método estático `replaceIgnoreCase` na classe `StringUtils` que receba uma string e dois caracteres como argumentos, e que faça a troca de caracteres maiúsculos e minúsculos simultaneamente, retornando a string modificada. Por exemplo, o resultado da execução do método quando passarmos como argumentos a string `mágica` e os caracteres `'a'` e `'o'` deve ser `"ObrocodobrO"`.

**Exercício 12.43:** ⋆⋆

Os métodos `startsWith` e `endsWith` da classe `String` consideram caracteres maiúsculos e minúsculos como sendo diferentes: se a string `cidade` valer `"Rio de Janeiro"` o resultado de `cidade.startsWith("rio")` será `false`.. Escreva dois métodos estáticos (`startsWithIgnoreCase` e `endsWithIgnoreCase`) na classe `StringUtils` que recebam duas strings como argumentos e que retornem `true` se a primeira string respectivamente começar ou terminar com a segunda, independentemente de estarem em maiúsculas ou minúsculas.

**Exercício 12.44:** ⋆⋆

Os métodos `indexOf` e `lastIndexOf` da classe `String` consideram caracteres maiúsculos e minúsculos como sendo diferentes: se a string `meses` valer `"agosto,setembro,outubro"` o resultado de `meses.indexOf("SETEMBRO")` será $-1$. Escreva dois métodos estáticos (`indexOfIgnoreCase` e `lastIndexOfIgnoreCase`) na classe `StringUtils` que recebam duas strings como argumentos e que retornem respectivamente a primeira e a última posição em que a segunda string ocorre na primeira, independentemente de estarem em maiúsculas ou minúsculas, ou o valor $-1$ se a segunda string não for encontrada na primeira.

**Exercício 12.45:** ⋆⋆

Os métodos `indexOf` e `lastIndexOf` da classe `String` consideram caracteres acentuados como sendo diferentes dos caracteres não-acentuados correspondentes. Escreva dois métodos estáticos (`indexOfIgnoreAcentos` e `lastIndexOfIgnoreAcentos`) na classe `StringUtils` que recebam duas strings como argumentos e que retornem respectivamente a primeira e a última posição em que a segunda string ocorre na primeira, independentemente de estarem acentuadas ou não, ou o valor $-1$ se a segunda string não for encontrada na primeira. Veja o exercício 12.21, que pode ajudar na solução deste problema.

**Exercício 12.46:** ⋆⋆

Escreva um método `retiraVogais` na classe `StringUtils` que receba uma string como argumento e remova todas as vogais (maiúsculas e minúsculas) desta string, retornando a string modificada como resultado.

**Exercício 12.47:** ⋆⋆

Modifique a classe `EpocaDeFrutas` (listagem 12.7) para que a comparação do mês passado como argumento com os meses nos quais as frutas possam ser colhidas seja feita considerando que caracteres maiúsculos e minúsculos são iguais. *Dica:* use o exercício 12.44.

**Exercício 12.48:** ⋆⋆

Usando o exercício 12.47 como base, faça com que a comparação também considere caracteres acentuados e não acentuados como sendo iguais (isto é, considerando que `'ã'` e `'a'` são iguais). *Dica:* use o exercício 12.21.

**Exercício 12.49:** ⋆⋆

Escreva um método `listaTerminais` na classe `StringUtils` (listagem 12.8) que receba uma string e um caracter como argumento, e imprima todas as substrings da string passada que terminem com o caracter passado. Por exemplo, se a string `"indeterminadamente"` e o caracter `'e'` forem passados como argumentos, o método deverá imprimir as strings `"inde"`, `"indete"`, `"indeterminadame"`, `"indeterminadamente"`.

**Exercício 12.50:** ⋆⋆

Escreva um método `quantasVezes` na classe `StringUtils` que receba duas strings como argumentos e retorne o número de vezes que a segunda string aparece na primeira. Por exemplo, se a string `"recrearem"` e `"re"` forem passadas como argumentos, o método deverá retornar 3.

**Exercício 12.51:** ⋆⋆

Escreva, para a classe `URL` (listagem 12.5), um método `retornaProtocolo` que retorne o protocolo da URL encapsulada. Por exemplo, se a URL encapsulada for `"http://java.sun.com"` o método deverá retornar `"http"`, e se a URL encapsulada for `"ftp://ftp.cdrom.com"` o método deverá retornar `"ftp"`. Se não houver protocolo aparente, como por exemplo na URL `"www.sun.com"`, o método deverá retornar `"http"`. *Dica:* existem ao duas maneiras de resolver este problema, considere a que parece ser mais flexível e completa.

**Exercício 12.52:** ⋆⋆⋆

Escreva uma classe `ArrayDeStrings`, no molde da classe `ArrayDeFloats` (listagem 11.4) que encapsule um array de strings. Esta classe deve ter os seguintes métodos:

- Construtor, que recebe como argumento um número máximo de instâncias da classe `String` que serão encapsuladas pela classe,
- `tamanho`, que retorna o número de strings no array encapsulado,
- `modifica`, que recebe como argumentos um valor inteiro (posição) e uma string, e faz com que a string naquela posição do array passe a ser a passada como argumento,
- `valor`, que recebe como argumento um valor inteiro (posição) e retorna a string armazenada naquela posição do array,
- `toString`, que retorna uma única string contendo todas as strings encapsuladas na classe, separadas por quebra de linhas (caracter `\n`).

**Exercício 12.53:** ⋆⋆⋆

Escreva, para a classe `ArrayDeStrings` (exercício 12.52), um método `filtra` que recebe como argumento uma string e retorna uma nova instância da classe `ArrayDeStrings` contendo todas as strings que contém a string passada como argumento. Por exemplo, se uma instância da classe contiver as strings [`"antialérgico"`, `"antigo"`, `"anticorpo"`, `"antilhana"`, `"antibiótico"`] e executarmos o método `filtra` passando como argumento a string `"co"`, o método deverá retornar uma nova instância de `ArrayDeStrings` contendo [`"antialérgico"`, `"anticorpo"`, `"antibiótico"`]. Escreva também um método `filtraIgnoreCase` que execute a mesma tarefa mas considerando que caracteres maiúsculos e minúsculos são iguais.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.54:** ⋆⋆⋆

Escreva, para a classe `ArrayDeStrings`, um método `existe` que receba como argumento uma string e retorne `true` caso a string passada como argumento exista (exatamente igual) no array de strings encapsulado pela classe. Escreva também um método `existeIgnoreCase` que execute a mesma tarefa mas considerando que caracteres maiúsculos e minúsculos são iguais.

**Exercício 12.55:** ⋆⋆⋆

Uma string é dita *palíndroma* se ela pode ser lida da esquerda para a direita ou da direita para a esquerda da mesma forma. As strings *"radar"*, *"asa"* e *"O breve verbo"* são palíndromas (desconsiderando os espaços). Escreva dois métodos estáticos na classe `StringUtils` (listagem 12.8) que retornem `true` se uma string passada como argumento for palíndroma e `false` se não for. Um dos métodos deve ser estrito e considerar espaços como caracteres, o outro não - como diferenciar os dois ? *Dica:* use o exercício 12.32 como base.

**Exercício 12.56:** ⋆⋆⋆

Escreva um método estático `removeCaracteres` na classe `StringUtils` que, recebendo duas strings como argumentos, retorne uma nova string que é a subtração dos caracteres da segunda string passada como argumento dos caracteres da primeira string. Por exemplo, se as strings `"Abracadabra"` e `"cabra"` forem passadas como argumentos, o algoritmo deve remover **uma** de cada uma das letras de `"cabra"` da palavra `"Abracadabra"`, resultando em `"Adabra"`. Se algum caracter da segunda string não existir na primeira, o método deve retornar uma string vazia (exemplo: se a segunda string for `"praxe"` e a primeira for `"paralelepípedo"`, a subtração não poderá ser efetuada.)

**Exercício 12.57:** ⋆⋆⋆

Escreva uma classe `StringDNA` que seja capaz de processar uma string de DNA. Strings de DNA são strings que são formadas exclusivamente pelos caracteres `'A'`, `'C'`, `'G'` e `'T'` - nenhum outro caracter é permitido. Esta classe deve encapsular uma instância da classe `String` e conter ao menos os seguintes métodos:

- Construtor, que recebe uma string normal como argumento e copia-a para a string encapsulada pela classe se a passada por argumento estiver dentro dos parâmetros exigidos (isto é, se a string passada for `"CATGATTAG"`, a string encapsulada será `"CATGATTAG"`, mas se a string passada for `"JAVA"`, a string encapsulada será `"AA"`.
- `toString`, que retorna a string encapsulada,
- `charAt`, que retorna o caracter na posição que for passada como argumento,
- `quantosA`, `quantosC`, `quantosG` e `quantosT`, que retornam, respectivamente, quantos caracteres `'A'`, `'C'`, `'G'` e `'T'` existem na string encapsulada,
- `length`, que retorna o comprimento da string encapsulada.

**Exercício 12.58:** ⋆⋆⋆

Uma string contendo RNA é composta somente dos caracteres `'A'`, `'C'`, `'G'` e `'U'` - nenhum outro caracter é permitido. Escreva uma classe `StringRNA` que herde da classe `StringDNA` (exercício 12.57) e implemente os mesmos métodos da classe `StringDNA`. Quais métodos devem ser sobreescritos ?

**Exercício 12.59:** ⋆⋆⋆

Escreva, para a classe `StringDNA` (exercício 12.57), um método `reversoComplementar` que retorne o reverso complementar da string encapsulada pela classe. O reverso complementar é calculado em dois passos: primeiramente trocamos cada caracter por seu complementar (isto é, `'A'` por `'T'` e vice-versa, `'C'` por `'G'` e vice-versa), depois revertemos a string de forma que o primeiro caracter seja o último e vice-versa. Por exemplo, se a string encapsulada for `"CTAGGATA"` o método deverá retornar `"TATCCTAG"`. O reverso complementar deve ser retornado como uma nova instância da própria classe `StringDNA`.

**Exercício 12.60:** ⋆⋆⋆

Crie, na classe `StringDNA` (veja exercício 12.57), o método `compara` que recebe uma instância da própria classe `StringDNA` para comparação e retorna um valor inteiro, calculado com o seguinte algoritmo:

- Coloque o valor zero em um acumulador,
- Para cada posição nas duas strings, compare os dois caracteres na posição,
- Se os caracteres forem exatamente iguais, some $+3$ pontos ao acumulador,
- Se os caracteres forem as combinações `'A'` e `'T'` **ou** `'T'` e `'A'`, some $+1$ ponto ao acumulador,
- Se os caracteres forem as combinações `'C'` e `'G'` **ou** `'G'` e `'C'`, some $+1$ ponto ao acumulador,
- Quando terminarem os caracteres de uma das strings, o valor acumulado será o valor a ser retornado pelo método.

Exemplo: Se a instância da classe `StringDNA` conter a string `"ACATTG"` e para o método `compara` for passada, como argumento, a string `"ATTCCG"`, o valor a ser retornado será $3+0+1+0+0+3=7$.

**Exercício 12.61:** ⋆⋆⋆

Modifique o exercício 12.29 para que nomes possam ser representados de maneira mais flexível, isto é, para que a classe possa representar nomes compostos de mais do que três componentes. Para isto, os métodos `rubrica` e `assinatura` devem ser modificados.

**Exercício 12.62:** ⋆⋆⋆

Escreva a classe `Criptografia`, que conterá alguns métodos estáticos para codificação e decodificação de strings. Escreva nesta classe o método `codificaRot13`, que receberá uma string como argumento e retornará uma string codificada com o algoritmo *rot13*, que substitui cada caracter da string pelo valor do caracter mais treze, subtraindo vinte e seis caso o resultado seja maior que a última letra, de forma que `"abCde"` seja substituída por `"noPqr"`, `"kLmnoPq"` seja substituída por `"xYzabCd"` e `"UVWxyz"` seja substituída por `"HIJklm"`. Somente os caracteres alfabéticos não acentuados devem ser modificados. Por exemplo, se a string `"Revolução de 1930"` for passada como argumento para este método, este retornará `"Eribyhçãb qr 1930"`. Uma característica interessante do algoritmo *rot13* é que se uma string codificada por ele for passada de novo pelo próprio algoritmo, a string original será retornada. Escreva também um método `decodificaRot13` que seja somente uma chamada para o método `codificaRot13`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.63:** ⋆⋆⋆

Escreva uma classe `CodigoMorse` com métodos estáticos que convertam strings de caracteres para strings em código morse e vice-versa. No código morse, caracteres são representados por pontos (correspondentes a um impulso elétrico curto) e traços (correspondentes a um impulso elétrico longo). Os caracteres básicos e seus correspondentes em código morse são mostrados abaixo:

| | | | | | |
|---|---|---|---|---|---|
| a .− | b −... | c −.−. | d −.. | e . | f ..−. |
| g −−. | h .... | i .. | j .−−− | k −.− | l .−.. |
| m −− | n −. | o −−− | p .−−. | q −−.− | r .−. |
| s ... | t − | u ..− | v ...− | w .−− | x −..− |
| y −.−− | z −−.. | | | ponto .−.−.− | vírgula −−..−− |

A cada caracter que for codificado, um espaço deve ser adicionado à string de saída. O método para codificação deve considerar caracteres maiúsculos e minúsculos como sendo equivalentes, e ignorar outros símbolos. Exemplo: se a string `"Farfalhar"` for passada para o método de codificação, este deve retornar `"..−. .− .−. ..−. .− .−.. .... .− .−."`.
*Dica:* para a decodificação de código morse para caracteres, use os espaços que devem existir na string codificada como delimitadores dos caracteres do código morse. Se algum código morse não tiver caracter correspondente (por exemplo, `"−−−−−"`), use o caracter `?` para saída.

**Exercício 12.64:** ⋆⋆⋆

O *Algoritmo de César* de criptografia de strings é uma versão melhorada do algoritmo *rot13* (veja exercício 12.62): o seu funcionamento é o mesmo, só que em vez de substituir cada caracter por um caracter treze posições depois, o algoritmo de César recebe um valor chamado *chave*, e usa este valor como o número de posições que devem ser puladas para a criptografia. Por exemplo, se a chave for 1 o algoritmo pulará uma posição ao codificar as letras, então se a string passada for `"Java"`, o resultado será `"Kbwb"`. O algoritmo de decodificação deve receber a mesma chave só que deve substituir os caracteres da string por valores em posições anteriores. Escreva um método estático `codificaCésar` na classe `Criptografia` que implemente o Algoritmo de César, recebendo como argumentos uma string e uma chave (valor numérico) e retornando a string criptografada. Este método deve considerar que **somente** as letras não-acentuadas devem ser criptografadas, as letras acentuadas, números, espaços e outros símbolos devem continuar como estão. Escreva também o método `decodificaCésar`.
*Dica:* Para simplificar o algoritmo, considere que o valor da chave só pode estar entre 1 e 25. Existem ao menos duas maneiras de implementar este algoritmo.

**Exercício 12.65:** ⋆⋆⋆

Escreva, para a classe `StringUtils` (listagem 12.8) o método `éSubSequênciaDe` que recebe duas strings como argumentos e retorna `true` se a primeira string for subsequência da segunda, e `false` se não for. Uma string é considerada uma subsequência de outra se todos os caracteres da primeira string aparecerem em ordem na segunda. Por exemplo, considerando a string `palavra` valendo `"consciensiosamente"`, a string `"ciente"` seria subsequência de `palavra` enquando `"cimento"` não seria. Podemos também definir que uma string `a` é uma subsequência de outra string `b` se pudermos formar `a` retirando caracteres de `b` mas mantendo a ordem.

**Exercício 12.66:** ⋆⋆⋆

O *Algoritmo das fatias* de criptografia usa como entrada uma string e um valor numérico, sendo que este valor numérico (o número de fatias) deve ser bem menor que o tamanho da string. O algoritmo de codificação fatia a string de entrada, tomando caracteres de $N$ em $N$ posições, onde $N$ é o número de fatias, formando $N$ novas strings cada uma com o comprimento $M$ onde $M$ é o comprimento da string original dividido por $N$. As strings criadas assim são concatenadas, sendo o resultado da codificação da string original. Para decodificar uma string criptografada com este algoritmo, é necessário ter o valor numérico.
Por exemplo, para criptografar a string "`Programação em Java`" (19 caracteres) usando 4 como número de fatias, o primeiro passo seria completar a string de forma que tenha um número de caracteres múltiplo de 4, para "`Programação em Java `" (20 caracteres), para simplificar o algoritmo. Fatiando esta string pegando de quatro em quatro caracteres obtemos quatro novas strings:

```
''Prçea''
''raãmv''
''omo~a''
''ga~J~''
```

A string criptografada seria o resultado da concatenação destas strings, ou "`Prçearaãmvomo aga J `". Para decodificar esta string, basta repetir o processo de codificação mas usando $M$ como o númeo de fatias, obtendo as strings

```
''Prog''
''rama''
''ção~''
''em~J''
''ava~''
```

cuja concatenação resulta em "`Programação em Java `".
Escreva, na classe `Criptografia`, o método `codificaFatias` que recebe uma string e um valor numérico, codificando a string usando o valor e retornando a string criptografada. Escreva também o método `decodificaFatias` que deve fazer o processo reverso. *Dica:* o método `decodificaFatias` pode ser uma chamada para o método `codificaFatias` com o valor numérico adequado.

**Exercício 12.67:** ⋆⋆⋆

Escreva, para a classe `StringUtils`, o método `éSubSetDe` que recebe duas strings como argumentos e retorna `true` se a primeira string for subset da segunda, e `false` se não for. Uma string é considerada um subset de outra se todos os caracteres da primeira string aparecerem em qualquer ordem na segunda, mas sem repetir caracteres da segunda. Por exemplo, considerando a string `palavra` valendo "`consciensiosamente`", as string "`ciente`" e "`cimento`" seriam subsets de `palavra`, mas "`consomem`" não seria pois requer dois caracteres `'m'`, e a string `palavra` somente contém um.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.68:** ⋆⋆⋆

Escreva, para a classe `StringUtils` (listagem 12.8) o método `alfabetoCompleto` que recebe uma string como argumento e retorna outra string contendo o alfabeto completo da string passada como argumento. O alfabeto completo de uma string é o grupo de caracteres que aparecem na string, sem repetições (podendo ser mostrado ordenado ou não). Por exemplo, o alfabeto completo de "`desencontradamente`" poderia ser "`desncotram`", e o alfabeto completo de "`colina`" poderia ser a própria string "`colina`".

**Exercício 12.69:** ⋆⋆⋆

Escreva, para a classe `StringUtils` dois métodos que comparem strings ignorando espaços. Um dos métodos pode ser chamado `equalsIgnoreSpaces` e o outro `equalsIgnoreSpacesIgnoreCase`. Se as strings "`localidade`" e "`local idade`" forem comparadas com os métodos `equals` ou `equalsIgnoreCase` elas serão diferentes, mas com os métodos `equalsIgnoreSpaces` e o outro `equalsIgnoreSpacesIgnoreCase` seriam consideradas iguais.

**Exercício 12.70:** ⋆⋆⋆

Um dos problemas com instâncias da classe `StringTokenizer` é que não podemos verificar que *tokens* já foram extraídos - só existem mecanismos para recuperar o próximo *token*, mas não para recuperar *tokens* anteriores. Crie uma classe `StringTokenizerComoArray` que encapsule um array de strings que seja obtido através de uma instância da classe `StringTokenizer`. O construtor desta classe deverá alocar e inicializar o array com os *tokens* de uma string passada como argumento, e a classe deve ter métodos que permitam a obtenção de qualquer *token*. *Dica:* veja a classe `ArrayDeFloats` (listagem 11.4).

**Exercício 12.71:** ⋆⋆⋆

O mecanismo de casamento de padrões mostrado na classe `CasamentoDePadroes` (listagem 12.10) considera caracteres maiúsculos e minúsculos como sendo diferentes. Modifique a classe `CasamentoDePadroes` de forma que esta encapsule também um valor booleano `ignoreCase`, que deve ser inicializado pelo construtor, usando um argumento. Se este valor booleano for igual a `true`, o método `casa` deverá considerar caracteres maiúsculos iguais a minúsculos. *Dica:* a resposta do exercício 12.43 poderá ser de ajuda.

**Exercício 12.72:** ⋆⋆⋆

Escreva um método na classe `CasamentoDePadroes` que calcule e retorne quantas strings diferentes podem ser formadas com o padrão encapsulado. Como referência, considere que o padrão "`1,2,3,4/8,9,10/2001`" pode formar 12 strings diferentes.

**Exercício 12.73:** ⋆⋆⋆

Usando o exercício 12.72 como base, escreva um método que imprima todas as strings diferentes que podem ser formadas com o padrão encapsulado.

**Exercício 12.74:** ⋆⋆⋆

Usando o exercício 12.73 como base, escreva um método que calcule e imprima o comprimento máximo e mínimo das strings que podem ser formadas com o padrão.

**Exercício 12.75:** ⋆⋆⋆

Modifique o construtor da classe `CasamentoDePadroes` (listagem 12.10) de forma que este verifique se o padrão pode ser analisado sem problemas. O padrão deve ter, para cada caracter `'{'`, um caracter `'}'` correspondente, e não deve ter pares de caracteres `'{'` e `'}'` aninhados.

**Exercício 12.76:** ⋆⋆⋆

Escreva uma aplicação em Java que leia uma string do teclado contendo algumas operações matemáticas simples, e que analise e calcule o resultado destas operações. A string conterá vários valores numéricos separados por sinais `'+'` e `'-'`, e deve considerar que valores sem sinais associados usam os últimos sinais que apareceram na string. Um exemplo de string que serviria de entrada para a aplicação é `"+ 5 4 3 - 9 2 + 22"`, que deve ser avaliada como $+5+4+3-9-2+20$, dando como resultado 21. *Dica:* veja a classe `CalculadoraDeLinhaDeComando` (listagem 11.12).

**Exercício 12.77:** ⋆⋆⋆

O *Algoritmo das pontas* de criptografia recebe uma string como argumento e produz uma outra string como resultado, e pode ser descrito da seguinte forma: enquanto a string de entrada conter caracteres, remova o primeiro e o último caracteres da string de entrada e os coloque na string de saída. Desta forma, se a string `"Programação em Java"` for entrada no algoritmo, este mostrará como saída a string `"ParvoagJr ammea çoã"`. A decodificação de uma string pode ser feita da seguinte forma: crie duas strings temporárias, e para cada par de caracteres extraídos da string codificada de entrada adicione o primeiro no fim da primeira string e o segundo no início da segunda string. A concatenação das duas strings é o resultado da decodificação.
Escreva, na classe `Criptografia`, os métodos estáticos `codificaPontas` e `decodificaPontas` para codificar e decodificar uma string usando este algoritmo.

**Exercício 12.78:** ⋆⋆⋆

Escreva a classe `EstatisticaDeCaracteres` que armazene contagem de caracteres em strings. Esta classe deve encapsular um array de 38 posições de valores inteiros, correspondentes aos caracteres maiúsculos `'A'` a `'Z'`, mais os dígitos `'0'` a `'9'`, mais o espaço `' '` e mais uma categoria "outros". A classe deve ter os seguintes métodos:

- Construtor, que inicializa o array com zeros,
- `processa`, que recebe como argumento uma string, converte-a para caracteres maiúsculos e processa os caracteres desta string, um a um, incrementando o elemento do array correspondente ao caracter,
- `valor`, que recebe como argumento um caracter e retorna o valor do array correspondente àquele caracter (use `'?'` para indicar a categoria "outros"),
- `toString`, que retorna uma string formatada com estatísticas sobre os caracteres processados.

Por exemplo, se uma instância desta classe for construída, o método `processa` for chamado passando como argumento a string `"Julho de 2001 (inverno)"`, e ao final do processamento o método `valor` for chamado várias vezes com os argumentos `'0'`, `'X'` e `'?'`, este deverá retornar respectivamente 2, 0 e 2 (correspondendo aos parênteses).

**Exercício 12.79:** ★★★

Escreva uma classe `FormataHTML` com um método estático `formataFórmulaQuímica` que receba como argumento uma string contendo uma fórmula química do tipo `"CH3CH2CH3"` e converta-a para um trecho de código HTML que faça com que os números apareçam em subscrito (em nível ligeiramente inferior), retornando o resultado desta conversão. Para que uma região de texto em HTML apareça em subscrito, ela deve estar entre os delimitadores `"<sub>"` e `"</sub>"`. Por exemplo, se este método receber como argumento `"CH3CH2CH3"` ele deverá retornar `"CH<sub>3</sub>CH<sub>2</sub>CH<sub>3</sub>"`.

**Exercício 12.80:** ★★★★

Escreva a classe `CasamentoDePadroesComCoringa` que execute o algoritmo de casamento de padrões mostrado na seção 12.4 mas considerando que o caracter `'?'` também possa ser usado, e que o algoritmo de casamento considere que este caracter é igual a qualquer outro caracter que apareca na string sendo analisada. Desta forma, se o padrão encapsulado for `"re,di??de"` as strings `"reside"`, `"divide"` e `"revide"` casarão com o padrão, enquanto as strings `"rebelde"` e `"relate"` não casarão.
*Dicas:* considere as diferenças do algoritmo no tratamento de caracteres `'?'` como sendo um *singleton* e como sendo parte da lista de alternativas. Vale a pena escrever esta classe como sendo herdeira da classe `CasamentoDePadroes` ? Quais as vantagens e dificuldades esperadas ?

**Exercício 12.81:** ★★★★

Escreva, para a classe `ArrayDeStrings` (exercício 12.52), um método `união` que receba, como argumento, uma outra instância da classe `ArrayDeStrings` e retorne uma nova instância da classe contendo a união das strings do array encapsulado com as do passado como argumento, sem repetições. Por exemplo, se o array encapsulado contiver as strings [`"célula"`, `"núcleo"`, `"plasma"`, `"mitocôndria"`] e o array passado como argumento para o método contiver [`"átomo"`, `"núcleo"`, `"plasma"`, `"molécula"`], o método deverá retornar o array [`"célula"`, `"núcleo"`, `"plasma"`, `"mitocôndria"`, `"átomo"`, `"molécula"`]. Escreva também o método `uniãoIgnoreCase`, que considera caracteres maiúsculos como sendo iguais aos minúsculos.

**Exercício 12.82:** ★★★★

Escreva, para a classe `ArrayDeStrings` (exercício 12.52), um método `intersecção` que receba, como argumento, uma outra instância da classe `ArrayDeStrings` e retorne uma nova instância da classe contendo a intersecção das strings do array encapsulado com as do passado como argumento. Por exemplo, se o array encapsulado contiver as strings [`"célula"`, `"núcleo"`, `"plasma"`, `"mitocôndria"`] e o array passado como argumento para o método contiver [`"átomo"`, `"núcleo"`, `"plasma"`, `"molécula"`], o método deverá retornar o array [`"núcleo"`, `"plasma"`]. Escreva também o método `intersecçãoIgnoreCase`, que considera caracteres maiúsculos como sendo iguais aos minúsculos.

**Exercício 12.83:** ★★★★

Aminoácidos são definidos por conjuntos de três caracteres em strings de RNA, sendo que cada aminoácido pode ter mais do que um conjunto de três caracteres correspondentes (*codons*). Existem vinte aminoácidos, e algumas combinações de três caracteres formam um *sinal de término*. Os vinte aminoácidos e o sinal de término, seus símbolos (entre parênteses) e as combinações correspondentes são:

- Ácido Aspártico (`D`): `GAU` e `GAC`
- Ácido Glutâmico (`E`): `GAA` e `GAG`
- Alanina (`A`): `GCU`, `GCC`, `GCA` e `GCG`
- Arginina (`R`): `CGU`, `CGC`, `CGA`, `CGG`, `AGA` e `AGG`
- Asparagina (`N`): `AAU` e `AAC`
- Cisteína (`C`): `UGU` e `UGC`
- Fenilalanina (`F`): `UUU` e `UUC`
- Glicina (`G`): `GGU`, `GGC`, `GGA` e `GGG`
- Glutamina (`Q`): `CAA` e `CAG`
- Histidina (`H`): `CAU` e `CAC`
- Isoleucina (`I`): `AUU`, `AUC` e `AUA`
- Leucina (`L`): `UUA`, `UUG`, `CUU`, `CUC`, `CUA` e `CUG`
- Lisina (`K`): `AAA` e `AAG`
- Metionina (`M`): `AUG`
- Prolina (`P`): `CCU`, `CCC`, `CCA` e `CCG`
- Serina (`S`): `AGU`, `AGC`, `UCU`, `UCC`, `UCA` e `UCG`
- Tirosina (`X`): `UAU` e `UAC`
- Treonina (`T`): `ACU`, `ACC`, `ACA` e `ACG`
- Triptofano (`W`): `UGG`
- Valina (`V`): `GUU`, `GUC`, `GUA` e `GUG`
- Sinais de término (`.`): `UAA`, `UAG` e `UGA`

Considerando a lista acima, escreva a classe `StringAminoAcidos`, que encapsule uma string composta somente de símbolos de aminoácidos. O construtor desta classe deve receber como argumento uma instância da classe `StringRNA` (exercício 12.58) e transformar grupos de três em três caracteres para símbolos dos aminoácidos, armazenando estes na string encapsulada. Por exemplo, se a string encapsulada por uma instância da classe `StringRNA` fosse `"AUGGGUAAAGCCUGGUAG"` e esta string fosse passada como argumento para o construtor da classe `StringAminoAcidos`, a string encapsulada seria `"MGKAW."`. O método deve desconsiderar restos de strings que não formem três caracteres: uma string de oito caracteres corresponderá à dois aminoácidos e dois caracteres sobrarão, sendo descartados.
*Dica:* existe mais de uma maneira de calcular o aminoácido a partir das trincas de caracteres. Qual é a mais simples ?



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.84:** ⋆⋆⋆⋆

Escreva, para a classe `ArrayDeStrings`, um método `ordenaCrescente` que ordene as strings contidas no array encapsulado em ordem crescente (alfabética) usando o algoritmo *bubblesort* e retorne o array ordenado. Escreva também um método `ordenaDecrescente` que ordene as strings em ordem decrescente. *Dica:* veja o exercício 11.111, use o método `compareTo` da classe `String` para comparar as strings para determinar a troca de posições.

**Exercício 12.85:** ⋆⋆⋆⋆

Escreva, para a classe `CasamentoDePadroes`, um método `casaComPesos` que ao invés de retornar `true` ou `false` caso o casamento da string com o padrão ocorra ou não, retorne um valor inteiro calculado da seguinte forma:

- Inicialmente (antes do casamento), o valor deve ser inicializado com zero.
- Se estivermos comparando um *singleton* com a string, e o resultado for igual, o valor deverá ser acrescido de duas vezes o comprimento do *singleton*.
- Se estivermos comparando um *singleton* com a string, e o resultado for diferente, o valor deverá ser decrescido de duas vezes o comprimento do *singleton*.
- Se estivermos comparando um dos *tokens* da lista de alternativas, e o resultado for igual, o valor deverá ser acrescido do comprimento do *token*.
- Se estivermos comparando um dos *tokens* da lista de alternativas, e o resultado for diferente, o valor deverá ser decrescido do comprimento do *token*.
- Ao final da análise dos *tokens*, o valor deverá ser decrescido de dois vezes cada caracter que ainda restar na string.

Escreva também uma classe que demonstre este novo algoritmo.

**Exercício 12.86:** ⋆⋆⋆⋆

Várias sequências de aminoácidos podem ser comparadas para obtenção de um consenso, que pode ser calculado como o caracter que mais aparece em uma determinada posição. Por exemplo, considerando as seguintes sequências de caracteres

```
''FNTXSPRNCDE''
''FCXTSRNRPDE''
''NNTXSRPNCCE''
''FNTXSPXRNDE''
```

o consenso seria calculado para cada posição como sendo o caracter mais frequente, e o resultado seria a string "`FNTXS???CDE`". Como em três posições não houve consenso (houve empate entre os caracteres mais frequentes), consideramos o caracter na posição como sendo igual a `'?'`.
Escreva na classe `StringAminoAcidos` (exercício 12.83) vários métodos `calculaConsenso` sobrecarregados, que recebam como argumentos outras instâncias da classe `StringAminoAcidos` e retornem o valor do consenso da string encapsulada com as passadas como argumentos. *Dica:* a classe `EstatisticaDeCaracteres` (exercício 12.78) pode ser útil para a resolução deste exercício.

**Exercício 12.87:** ★★★★

Duas sequências de aminoácidos podem ser comparadas entre si, caracter a caracter, para verificar o seu alinhamento. Em um alinhamento ideal, todos os caracteres são iguais nas duas sequências, mas frequentemente algumas divergências existem. Para avaliar a qualidade do alinhamento, um sistema de pontos é usado, que dá diferentes pesos ou notas para diferentes alinhamentos. Estes sistemas de pontos envolvem matrizes de substituição, que contém valores (pesos) que serão usados quando o aminoácido da coluna da matriz for comparado com o aminoácido na linha da matriz. Uma destas matrizes de substituição, chamada BLOSUM62, é mostrada abaixo.

| | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | +4 | −1 | −2 | −2 | 0 | −1 | −1 | 0 | −2 | −1 | −1 | −1 | −1 | −2 | −1 | +1 | 0 | −3 | −2 | 0 |
| **R** | −1 | +5 | 0 | −2 | −3 | +1 | 0 | −2 | 0 | −3 | −2 | +2 | −1 | −3 | −2 | −1 | −1 | −3 | −2 | −3 |
| **N** | −2 | 0 | +6 | +1 | −3 | 0 | 0 | 0 | +1 | −3 | −3 | 0 | −2 | −3 | −2 | +1 | 0 | −4 | −2 | −3 |
| **D** | −2 | −2 | +1 | +6 | −3 | 0 | +2 | −1 | −1 | −3 | −4 | −1 | −3 | −3 | −1 | 0 | −1 | −4 | −3 | −3 |
| **C** | 0 | −3 | −3 | −3 | +9 | −3 | −4 | −3 | −3 | −1 | −1 | −3 | −1 | −2 | −3 | −1 | −1 | −2 | −2 | −1 |
| **Q** | −1 | +1 | 0 | 0 | −3 | +5 | +2 | −2 | 0 | −3 | −2 | +1 | 0 | −3 | −1 | 0 | −1 | −2 | −1 | −2 |
| **E** | −1 | 0 | 0 | +2 | −4 | +2 | +5 | −2 | 0 | −3 | −3 | +1 | −2 | −3 | −1 | 0 | −1 | −3 | −2 | −2 |
| **G** | 0 | −2 | 0 | −1 | −3 | −2 | −2 | +6 | −2 | −4 | −4 | −2 | −3 | −3 | −2 | 0 | −2 | −2 | −3 | −3 |
| **H** | −2 | 0 | +1 | −1 | −3 | 0 | 0 | −2 | +8 | −3 | −3 | −1 | −2 | −1 | −2 | −1 | −2 | −2 | +2 | −3 |
| **I** | −1 | −3 | −3 | −3 | −1 | −3 | −3 | −4 | −3 | +4 | +2 | −3 | +1 | 0 | −3 | −2 | −1 | −3 | −1 | +3 |
| **L** | −1 | −2 | −3 | −4 | −1 | −2 | −3 | −4 | −3 | +2 | +4 | −2 | +2 | 0 | −3 | −2 | −1 | −2 | −1 | +1 |
| **K** | −1 | +2 | 0 | −1 | −3 | +1 | +1 | −2 | −1 | −3 | −2 | +5 | −1 | −3 | −1 | 0 | −1 | −3 | −2 | −2 |
| **M** | −1 | −1 | −2 | −3 | −1 | 0 | −2 | −3 | −2 | +1 | +2 | −1 | +5 | 0 | −2 | −1 | −1 | −1 | −1 | +1 |
| **F** | −2 | −3 | −3 | −3 | −2 | −3 | −3 | −3 | −1 | 0 | 0 | −3 | 0 | +6 | −4 | −2 | −2 | +1 | +3 | −1 |
| **P** | −1 | −2 | −2 | −1 | −3 | −1 | −1 | −2 | −2 | −3 | −3 | −1 | −2 | −4 | +7 | −1 | −1 | −4 | −3 | −2 |
| **S** | +1 | −1 | +1 | 0 | −1 | 0 | 0 | 0 | −1 | −2 | −2 | 0 | −1 | −2 | −1 | +4 | +1 | −3 | −2 | −2 |
| **T** | 0 | −1 | 0 | −1 | −1 | −1 | −1 | −2 | −2 | −1 | −1 | −1 | −1 | −2 | −1 | +1 | +5 | −2 | −2 | 0 |
| **W** | −3 | −3 | −4 | −4 | −2 | −2 | −3 | −2 | −2 | −3 | −2 | −3 | −1 | +1 | −4 | −3 | −2 | +11 | +2 | −3 |
| **Y** | −2 | −2 | −2 | −3 | −2 | −1 | −2 | −3 | +2 | −1 | −1 | −2 | −1 | +3 | −3 | −2 | −2 | +2 | +7 | −1 |
| **V** | 0 | −3 | −3 | −3 | −1 | −2 | −2 | −3 | −3 | +3 | +1 | −2 | +1 | −1 | −2 | −2 | 0 | −3 | −1 | +4 |

Usando esta matriz, podemos calcular o alinhamento entre as sequências de aminoácidos "`TKVSRYV`" e "`TDVAYYL`" como sendo a soma dos coeficientes mostrados na matriz: $+5-1+4+1-2+7+1$, ou 15.
Escreva na classe `StringAminoAcidos` (exercício 12.83) um método `calculaAlinhamento` que receba como argumento outra instância da classe `StringAminoAcidos` e retorne o valor do alinhamento da string encapsulada com a passada como argumento.

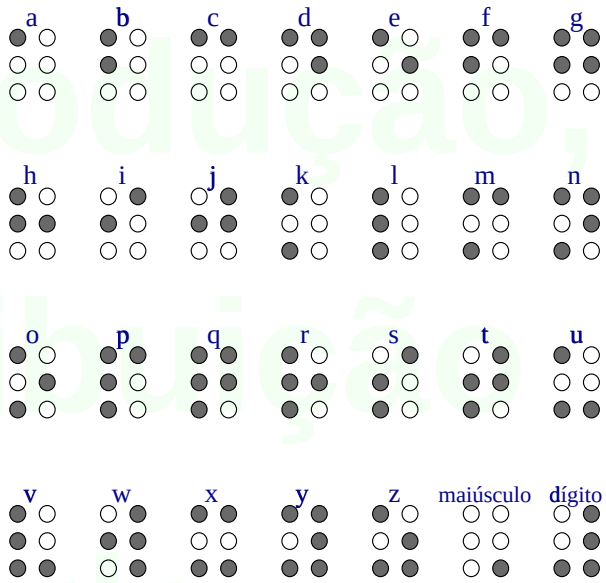
**Exercício 12.88:** ★★★★

Uma palavra ou frase é dita *anagrama* de outra se ela pode ser formada com os caracteres de outra, sem repetições, modificando as posições e inserindo espaços e pontuação à vontade, valendo também transformar maiúsculas em minúsculas e vice-versa. Desta forma, "`manda jogar em vapor`" é anagrama de "`programando em java`". Anagramas são curiosidades linguísticas, geralmente sendo feitos para satirizar nomes de pessoas ou locais. Para que um anagrama seja considerado interessante, as palavras formadas devem ter algum significado. Escreva, para a classe `StringUtils` (listagem 12.8), um método `éAnagrama` que receba duas strings como argumentos e retorne `true` se uma é anagrama de outra.

**Exercício 12.89:** ★★★★

Escreva a classe `StringBraille` em Java. Esta classe deve representar internamente uma string e ser capaz de imprimi-la no alfabeto braille, devendo ter ao menos o construtor (que recebe uma string a ser encapsulada, como argumento) e o método `toString` que imprimirá a string encapsulada em braille.

A figura abaixo mostra o alfabeto braille simplificado, onde um círculo preenchido significa uma marca em relevo no papel. Cada letra maiúscula que aparecer no texto deve ser precedida pelo caracter `maiúsculo` do alfabeto braille. Cada dígito que aparecer no texto deve ser precedido pelo caracter `dígito` do alfabeto braille. No caso dos dígitos, os caracteres braille correspondentes a `'a'`, `'b'`, `'c'` ... `'i'`, `'j'` são usados para representar os dígitos `'1'`, `'2'`, `'3'` ... `'9'`, `'0'`. Para simplificar, considere que as strings a serem convertidas não contém acentos nem símbolos, e que um espaço em braille pode ser representado por um caracter braille sem nenhuma marca em relevo.




A saída do programa pode ser feita usando os caracteres de texto `'X'` para representar uma marca em relevo, `'.'` para representar uma posição onde não há marca em relevo, e o espaço para separar uma letra do alfabeto braille de outra. Assim, se a string `"Java 123"` for entrada, a saída deverá ser:

```
.. .X X. X. X. .. .X X. .X X. .X XX
.. XX .. X. .. .. .X .. .X X. .X ..
.X .. .. XX .. .. XX .. XX .. XX ..
```

*Dica:* O método `toString` desta classe deve criar três strings na saída, cada uma com uma "linha" de pontos dos caracteres braille. O comprimento destas três strings é igual, mas deve ser calculado verificando-se se a string encapsulada tem caracteres maiúsculos e dígitos.

**Exercício 12.90:** ★★★★

Escreva, para a classe `StringBraille` (exercício 12.89), um método `decodifica` que recebe três strings contendo os caracteres 'X', '.' e espaço e decodifique a mensagem em Braille contida nestas strings para uma string comum. Por exemplo, se as três strings passadas forem respectivamente `".. .X X. X. X. .. .X X. .X X. .X XX"`, `".. XX .. X. .. .. .X .. .X X. .X .."` e `".X .. .. XX .. .. XX .. XX .. XX .."` o método deverá retornar a string `"Java 123"`. Veja o exemplo dado no exercício 12.89.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.91:** ★★★★

O *Algoritmo de César* (veja o exercício 12.64) pode ser implementado de maneira mais complexa (e difícil de ser quebrada), se ao invés de uma única chave, várias forem usadas. O primeiro caracter da string será codificado com a primeira chave, o segundo caracter com a segunda chave, etc. Quando as chaves acabarem, a primeira será reutilizada, até o final da string a ser codificada. A chave pode ser especificada por outra string, onde cada caracter tem um valor numérico correspondente, de forma que a string `"AEIY"` corresponde a quatro chaves (`1`, `5`, `9`, `25`). Desta forma, se a string `"Programa"` fosse codificada com a chave `"aeb"` (correspondente aos valores 1, 5 e 2, o resultado seria `"Qwqhwcnf"` (`'P'` foi adicionada a 1 e o resultado é `'Q'`, `'r'` foi adicionada a 5 e o resultado é `'w'`, `'o'` foi adicionada a 2 e o resultado é `'q'`, `'g'` foi adicionada a 1 e o resultado é `'h'`, `'r'` foi adicionada a 5 e o resultado é `'w'`, etc.) - notem o uso cíclico dos valores da chave.
Escreva um método estático `codificaCésar` na classe `Criptografia` (veja exercício 12.62) que implemente o Algoritmo de César modificado, recebendo duas strings como argumento: a primeira será a string a ser criptografada e a segunda será a chave. Este método deverá retornar outra instância da classe `String` correspondendo ao primeiro argumento, criptografado. Este método deve considerar que **somente** as letras não-acentuadas devem ser criptografadas, as letras acentuadas, números, espaços e outros símbolos devem continuar como estão. Escreva também o método `decodificaCésar` que também recebe duas strings como argumentos e retorna a string decodificada.
*Dica:* Para simplificar o algoritmo, considere que o valor da chave só pode estar entre 1 e 26, ou seja, as letras `'A'` a `'Z'`. O algoritmo de codificação deve verificar se a string passada como chave é válida, ou seja, se ela contém somente caracteres maiúsculos na faixa prevista.

**Exercício 12.92:** ★★★★★

Escreva uma classe `JogoSegueLetras` que encapsule array bidimensional de caracteres. Este array pode ser passado como argumento para o construtor da classe ou criado de outra forma. Para simplificar, considere que os caracteres armazenados serão todos maiúsculos (escreva código no construtor que garanta isto). Escreva para esta classe um método `existe` que receba uma string como argumento e que retorne um valor inteiro.
Uma string existe no array se é possível criar esta string navegando-se no array, um caracter de cada vez, sendo que de um caracter só se pode ir para o próximo se este for vizinho do caracter anterior. Se a string for encontrada, o valor retornado será o comprimento da string. Se a string não for encontrada, o valor retornado deverá ser menos duas vezes o comprimento da string. Por exemplo, se o array encapsulado for o mostrado abaixo, as palavras `"TIRO"`, `"DELTA"`, `"TALENTO"` e `"MAJORITARIAMENTE"` poderão ser achadas no array, e deverão retornar os valores 4, 5, 7 e 16, respectivamente, mas a palavra `"DESPROPORCIONADAMENTE"` deverá retornar o valor $-42$.

```
IRO
TAJ
LMD
SEO
RNI
OTE
```

**Exercício 12.93:** ★★★★★

Escreva uma classe `StringCircular` que represente uma string circular. Uma string circular é uma string onde considera-se que após o último caracter, o primeiro caracter da string é repetido, e antes do primeiro caracter, o último caracter da string aparece. Uma string circular é representada internamente por uma string normal, mas os métodos que acessam esta string devem ser capazes de processá-la como se fosse circular e infinita. Uma string circular não deveria ter um caracter inicial, mas para referência consideramos o primeiro caracter da string encapsulada como sendo o primeiro caracter da string circular.
Escreva para esta classe os seguintes métodos:

- Construtor, que recebe uma string normal como argumento e copia-a para a string encapsulada pela classe.
- `charAt`, que retorna o caracter na posição que for passada como argumento. Em uma string circular, o argumento do método `charAt` pode assumir valores maiores do que o comprimento real da string encapsulada ou mesmo menores do que zero. Por exemplo, se a string encapsulada for `"asterix"`, o método `charAt(10)` deve retornar o caracter `'e'` e o método `charAt(-1)` deve retornar o caracter `'x'`. O operador módulo (`%`) será bastante útil para a implementação do método `charAt`.
- `equals`, em duas versões sobrecarregadas: uma recebe uma instância da classe `String` como argumento, e outro que recebe uma instância da própria classe `StringCircular` e verifica se são iguais. A comparação de strings quando uma é circular deve levar a circularidade em conta: as strings `"asterix"` e `"rixaste"` são iguais se uma ou duas delas forem circulares. Uma solução simples para a implementação deste método pode ser feita se uma string temporária, contendo várias repetições da string encapsulada, for usada.
- `equalsIgnoreCase`, igual ao método `equals` descrito acima, exceto que caracteres maiúsculos e minúsculos devem ser considerados como iguais.
- `indexOf`, que retorna a primeira ocorrência da string passada como argumento. Por exemplo, se a string encapsulada for `"asterix"`, `indexOf("xas")` deve retornar 6 (por causa da circularidade) e `indexOf("xx")` deve retornar -1 pois a string passada como argumento não foi encontrada.

**Exercício 12.94:** ★★★★★

Escreva uma classe `JogoLocalizaPalavras` que encapsule um array bidimensional de caracteres. Este array pode ser passado como argumento para o construtor da classe ou criado de outra forma. Para simplificar, considere que os caracteres armazenados serão todos maiúsculos (escreva código no construtor que garanta isto). Escreva para esta classe um método `localiza` que receba uma string como argumento e retorne `true` caso esta string possa ser localizada dentro do array de caracteres, em qualquer posição e em qualquer orientação. Desta forma, caso o array encapsulado seja como o mostrado abaixo, as palavras "`LOCALIZAR`", "`TENTATIVA`", "`BAIXA`" e "`TESTE`" poderiam ser localizadas no array.

```
ABLNHEHLLTBQJFRGQH
KJUTWRAZILACOLVMNJ
FEOGEQHTLOIDFMBAOQ
RWBNUSGEVIXOIOXGUZ
BRDARGTENTATIVAYJK
EARHSOWESLFVCDPZJQ
WECSWATLXBMTLCDPNI
```

## 12.6 Exercícios complementares do capítulo 12

**Exercício 12.95:** ⋆

Modifique a classe `DemoJogoDaForca` (listagem 12.2) para que a palavra a ser adivinhada seja passada pela linha de comando.

**Exercício 12.96:** ⋆

A forma na qual o jogo da forca é implementada pela classse `DemoJogoDaForca` (listagem 12.2) permite que o jogador tente até acertar a palavra. Modifique o método `main` da classe `DemoJogoDaForca` para que somente um determinado número de tentativas possa ser feito. Este número pode ser calculado como $2c/3$, onde $c$ é o comprimento da palavra.

**Exercício 12.97:** ⋆

Explique por que a classe `StringUtils` (listagem 12.8) não pode ser declarada como herdeira da classe `String`.

**Exercício 12.98:** ⋆⋆

Modifique a classe `EpocaDeFrutas` (listagem 12.7) para que vários nomes de meses possam ser passados como argumentos pela linha de comando, e que o programa imprima todas as frutas que podem ser colhidas em qualquer um dos meses passados como argumentos.

**Exercício 12.99:** ⋆⋆

Usando a classe `EpocaDeFrutas` como base, escreva uma aplicação que mostre em que mês mais frutas podem ser colhidas.

**Exercício 12.100:** ⋆⋆

Escreva um construtor para a classe `Data` que receba uma string como argumento, e deduza os valores do dia, mês e ano a serem encapsulados a partir desta string. Esta string deve estar no formato `"DD/MM/AAAA"` - dois dígitos para o dia, uma barra, dois dígitos para o mês, outra barra, e quatro dígitos para o ano. Alguns exemplos válidos de argumentos para este construtor seriam `"01/01/0001"` e `"25/10/2000"`. *Dica:* veja como converter de strings para valores numéricos no apêndice A.

**Exercício 12.101:** ⋆⋆

Modifique o construtor do exercício 12.100 para que dois formatos de strings sejam aceitos: `"DD/MM/AAAA"` e `"DDMMAAAA"`. O construtor deve ser capaz de processar os dois formatos, determinando qual deve ser usado automaticamente. *Dica:* como não é possível usar a sobrecarga para criar dois construtores que recebem uma string como argumento, todo o processamento, inclusive a decisão sobre qual formato foi passado como argumento, deverá ser feito dentro do construtor.

**Exercício 12.102:** ⋆⋆

Escreva um construtor para a classe `ArrayDeFloats` (listagem 11.4) que receba como argumento uma única string contendo vários valores de ponto flutuante separados por espaços, e que inicialize os elementos do array com estes valores, com o processamento adequado.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 12.103:** ⋆⋆

Escreva um construtor para a classe `MatrizDeDoubles` (listagem 11.9) que receba como argumento uma única string contendo vários valores de ponto flutuante separados por espaços, precedidos de dois valores inteiros (as dimensões da matriz), e que inicialize os elementos da matriz com estes valores, com o processamento adequado. Um exemplo de string que poderia ser passada como argumento para este construtor é "`2 3 9.8 1.7 12 -48 -12 0`", representando uma matriz $2 \times 3$.

**Exercício 12.104:** ⋆⋆

Considere o construtor da classe `RegistroAcademico` (exercício 6.29). Faça uma outra versão deste construtor onde a informação sobre o curso é passada como uma string contendo o nome do curso. O construtor deverá, a partir da string passada como argumento para identificar o curso, reconhecer o código correspondente àquele curso. *Dica:* Tente escrever um método versátil, que reconheça tanto "`Engenharia da Computação`" quanto "`engenharia da computação`" quanto "`  engenharia da computação  `" como identificadores correspondentes ao código 39.

**Exercício 12.105:** ⋆⋆

Modifique o resultado do exercício 12.98 para que a classe verifique se os parâmetros `-help` ou `-h` foram passados pela linha de comando, e se tiverem sido passados, que imprima uma breve mensagem sobre o uso do programa, saindo em seguida.

**Exercício 12.106:** ⋆⋆

Existem vários pontos na classe `CasamentoDePadroes` (listagem 12.10) que podem ser otimizados para que o algoritmo de casamento seja mais eficiente. Um dos pontos é a eliminação da necessidade de recriar a instância da classe `StringTokenizer` para os tokens do padrão a cada chamada do método `casa` (veja os exercícios 12.70 e 12.72 para idéias de como eliminar esta necessidade). Você consegue encontrar outros pontos para otimização ?

**Exercício 12.107:** ⋆⋆⋆

Considere a classe `DemoLogin` (listagem 12.4), que verifica que nome de autor foi entrado e imprime uma frase deste autor. Escreva a classe `Citacoes` que encapsule, usando arrays de strings, autores e respectivas citações. Escreva nesta classe um método `menu` que mostre, em um menu, os nomes dos autores, permitindo ao usuário escolher um nome e ver a(s) frase(s) deste autor. Todos os métodos desta classe podem ser estáticos.

**Exercício 12.108:** ⋆⋆⋆

Escreva, para a classe `Citacoes`, um método estático `porTema` que receba como argumento uma string e que mostre todas as citações que contenham esta string. *Dica:* ao verificar se a string passada está contida em uma string contendo uma citação, é melhor ignorar diferenças entre maiúsculas e minúsculas e ignorar acentos. Veja os exercícios 12.44 e 12.45, que podem dar soluções para parte do problema.

# Capítulo 13

# Classes para Representação e Processamento de Valores Numéricos

## 13.1 Introdução

Computadores tem a capacidade de fazer processamento de dados numéricos com precisão e velocidade incomparáveis a de seres humanos[1]. Muitas das aplicações que usamos em computadores modernos usam extensivamente a capacidade destes de processamento numérico rápido e preciso, como em planilhas, softwares gráficos, aplicações financeiras e científicas e até mesmo em jogos.

A capacidade de processamento de valores numéricos (geralmente através de bibliotecas que contém as funções que efetuam os cálculos) é parte de qualquer linguagem de programação moderna. Neste capítulo veremos que classes a linguagem Java provê para processamento de valores numéricos, a sintaxe e regra de uso de seus métodos e algumas de suas aplicações. Veremos também alguns valores numéricos especiais que podem ser representados adequadamente por variáveis em Java, e as regras de uso destes valores especiais em operações matemáticas.

## 13.2 Valores numéricos especiais

*campo* `NaN`

Java provê campos especiais em algumas classes que são usados para representar valores especiais. Um destes valores é o já visto `NaN`, campo das classes `Float` e `Double`, que significa *Not a Number* (não é um número). Este valor é o resultado da divisão de zero por zero[2], e pode ser usado em operações matemáticas, embora qualquer operação que envolva o valor `NaN` resulte em `NaN`.

*campo* `POSITIVE_INFINITY`

As classes `Float` e `Double` também contém campos públicos, estáticos e finais para representar valores infinitos, positivos e negativos. Valores positivos infinitos são representados pelos campos `POSITIVE_INFINITY` e são o resultado da divisão de qualquer valor positivo por zero. Algu-

[1] Por mera curiosidade vale a pena a apresentação da seguinte frase: *"Multiplique na sua cabeça" (ordenou o Dr. Adams) "365.365.365.365.365.365 por 365.365.365.365.365.365". Ele (o garoto Truman Harry, de dez anos de idade) rodopiou em torno da sala como um pião, puxou suas calças compridas sobre o cano de suas botas, mordeu suas mãos, rolou os olhos, às vezes conversando e rindo, e então parecendo em agonia, até que em não mais do que um minuto, respondeu "133.491.850.208.566.925.016.658.299.941.583.255!". Um computador eletrônico poderia realizar a tarefa um pouco mais rápido, mas não seria tão divertido para se ver.* Frase de James R. Newman, autor de "O Mundo da Matemática", traduzida de uma mensagem do programa `fortune`.

[2] `NaN` também é resultado das operações $\infty/\infty$, $\infty-\infty$, $0\times\infty$ e $1^{\infty}$. Apesar das operações $0^0$ e $0^{\infty}$ serem consideradas matematicamente indefinidas, Java as considera como iguais a 1.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

mas operações que envolvem valores `POSITIVE_INFINITY` resultarão em `POSITIVE_INFINITY`.

*campo* `NEGATIVE_INFINITY`

Valores negativos infinitos são representados pelos campos `NEGATIVE_INFINITY` declarados nas classes `Float` e `Double` e são obtidos como resultados da divisão de qualquer valor negativo por zero.

*zero positivo*

*zero negativo*

Outros dois valores especiais são zeros positivo e negativo - embora as expressões "zero positivo" e "zero negativo" pareçam contra-senso, o valor zero positivo pode ser obtido pela divisão de um valor positivo qualquer por uma das constantes `POSITIVE_INFINITY` das classes `Float` ou `Double`. Similarmente, o valor zero negativo pode ser obtido pela divisão de um valor positivo qualquer por `Float.NEGATIVE_INFINITY` ou `Double.NEGATIVE_INFINITY`. Curiosamente, para comparações com o operador `==`, zeros positivos e negativos são considerados iguais. Mais detalhes sobre operações com os valores `NaN`, zeros e infinitos serão mostradas na seção A.2.7.

## 13.3 A classe `Math`

*classe* `Math`

A classe `Math`, parte do pacote `java.lang`, padrão de qualquer implementação da linguagem Java, contém alguns campos públicos e finais para a representação de constantes matemáticas e métodos estáticos para cálculos das funções matemáticas mais usadas. Estas constantes e funções e alguns exemplos de uso serão vistas nesta seção.

*classe* `StrictMath`

As versões de Java (1.3 e superior) também tem uma classe alternativa que contém constantes e métodos para processamento matemático, a classe `StrictMath`. Os resultados dos cálculos usando os métodos desta classe são garantidamente iguais independentemente da plataforma sendo usada, enquanto que os da classe `Math` podem ter implementações de desempenho melhor.

*campo* `E`

*campo* `PI`

As duas constantes matemáticas representadas pela classe `Math` são *e* (a base dos logaritmos naturais, aproximadamente igual a 2.718281828459045), que pode ser obtido usando-se a expressão `Math.E`, e o valor de $\pi$ (aproximadamente igual a 3.141592653589793), que pode ser obtido usando-se a expressão `Math.PI`.

### 13.3.1 Métodos de comparação e arredondamento

*método* `max`

*método* `min`

A classe `Math` provê alguns métodos que podem ser usados para comparação de dois valores. Estes métodos são sobrecarregados, para que possam ser chamados com vários tipos de dados diferentes. Um destes métodos é o método `max`, que retorna o maior de dois valores passados como argumentos, e pode ser chamado como `Math.max`. Existem quatro versões deste método, que aceitam argumentos e retornam valores dos tipos `int`, `long`, `float` e `double`. Similarmente, o método `min`, que pode ser executado como `Math.min`, retorna o menor dos dois valores passados como argumentos. Embora não existam formas destes métodos que recebam mais argumentos, eles podem ser facilmente combinados com chamada em cascata. Por exemplo, o resultado da expressão `Math.max(Math.max(a,b),Math.max(c,d))` seria o maior dos valores `a`, `b`, `c` e `d` (comparando primeiro `a` com `b` e `c` com `d` e finalmente os resultados das duas comparações).

*método* `rint`

A classe `Math` também contém vários métodos para arredondamento e cálculo do valor absoluto. O método `rint` recebe como argumento um valor do tipo `double` e retorna outro valor do tipo `double` que é matematicamente inteiro e que é o valor mais próximo do argumento. Se existirem dois inteiros que são equidistantes do valor passado como argumento, o valor que for par será retornado. Desta forma, o resultado de `Math.rint(10.8)` será 11.0, o resultado de `Math.rint(-10.2)` será $-10.0$ e o resultado de `Math.rint(10.5)` e de `Math.rint(9.5)` será



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

10.0. Se o valor passado como argumento for um valor matematicamente inteiro, ou se for igual a zero, infinito positivo ou negativo ou NaN, o método retornará o valor passado como argumento.

*método* round

O método round também arredonda o valor passado como argumento, mas de forma ligeiramente diferente do método rint. Este método tem duas versões, uma que recebe um argumento do tipo double e retorna um valor do tipo long, e outro que recebe como argumento um valor do tipo float e retorna um valor do tipo int. Ambos os métodos retornam o valor inteiro mais próximo do valor passado como argumento, com algumas exceções: se o argumento for NaN, o resultado será zero; se a parte inteira do argumento for maior do que o maior valor representável pelo tipo nativo, o valor máximo será retornado (Integer.MAX_VALUE ou Long.MAX_VALUE, dependendo da assinatura do método usado), e se a parte inteira do argumento for menor do que o menor valor representável pelo tipo nativo, o valor mínimo será retornado (Integer.MIN_VALUE ou Long.MIN_VALUE, dependendo da assinatura do método usado). Por exemplo, o resultado da expressão Math.round(1e18) é 1000000000000000000, enquanto que o resultado de Math.round(1e18f) é 2147483647 (como o valor passado foi do tipo float, o resultado deve ser do tipo int, e como este resultado seria maior que Integer.MAX_VALUE, o valor Integer.MIN_VALUE foi retornado). Outros exemplos são: Math.round(7.51) retornando 8 e Math.round(-7.51) retornando $-8$.

*método* ceil

Outro método de arredondamento é o método ceil, que espera como argumento um valor do tipo double e retorna outro valor do mesmo tipo, sendo este igual ao menor valor matematicamente inteiro que ao mesmo tempo não é menor que o valor passado como argumento, ou seja, o valor arredondado "para cima". Como exemplos da execução deste método podemos usar as expressões Math.ceil(200.8) (cujo resultado é 201.0), e Math.ceil(200.1) (cujo resultado também é 201.0). Se o argumento passado já for um valor matematicamente inteiro, ou se for igual a NaN ou infinito positivo ou negativo, o mesmo valor será retornado pelo método. É interessante observar que pela definição do método, qualquer valor que seja passado como argumento que esteja entre $-1$ e zero (exclusivo) será arredondado para $-0$, o que é um valor perfeitamente válido em Java.

*método* floor

O método floor de arredondamento funciona de maneira semelhante ao método ceil, mas retornando o maior valor matematicamente inteiro que não é maior que o argumento, ou seja, arredondando "para baixo". Desta forma, o resultado da expressões Math.floor(200.8) e Math.floor(200.1) é 200.0, o resultado de Math.floor(-100.9) é $-101.0$ e o resultado de Math.floor(-0.8) é $-1.0$. Se o argumento já for um valor matematicamente inteiro, NaN, infinito ou zero, o método retornará o mesmo valor do argumento.

*método* abs

O método abs não arredonda valores, mas retorna o valor absoluto (sem sinal) do argumento passado. Este método existe em quatro versões, que aceitam um valor dos tipos int, long, float ou double e retornam um valor do mesmo tipo sem o sinal - valores positivos são retornados como foram passados, e valores negativos são retornados como positivos. Alguns casos especiais existem: se o argumento for do tipo int ou long e for igual ao menor valor representável pelo tipo nativo (respectivamente Integer.MIN_VALUE ou Long.MIN_VALUE), este valor não será convertido para positivo, sendo retornado sem modificações, pois não é possível representar valores positivos iguais a -Integer.MIN_VALUE (com o tipo int) ou -Long.MIN_VALUE (com o tipo long).

### 13.3.2 Métodos trigonométricos

A classe `Math` também contém vários métodos para cálculo de funções trigonométricas simples. Como em outras linguagens de programação, os métodos que implementam funções trigonométricas manipulam valores em radianos.

*método* `sin` O primeiro método trigonométrico da classe `Math` é o método `sin`, que calcula e retorna o seno do ângulo passado como argumento para o método. Se o argumento passado for `NaN`, infinito positivo ou infinito negativo, o método retornará `NaN`. Se o argumento for zero positivo ou zero negativo, o argumento será retornado. Como exemplo, a expressão `Math.sin(Math.PI/6.)` será aproximadamente igual a 0.5.

*método* `cos` O método `cos` calcula e retorna o coseno do ângulo passado como argumento para o método. Se o argumento passado for `NaN`, infinito positivo ou infinito negativo, o método retornará `NaN`. Como exemplo, a expressão `Math.cos(Math.PI/4.)` será aproximadamente igual a $\sqrt{2}/2$.

*método* `tan` O método `tan` calcula e retorna a tangente do ângulo passado como argumento para o método. Se o argumento passado for `NaN`, infinito positivo ou infinito negativo, o método retornará `NaN`. Se o argumento for zero positivo ou zero negativo, o argumento será retornado. Como exemplo, a expressão `Math.tan(Math.PI)` será aproximadamente igual a zero.

*método* `asin` O método `asin` calcula e retorna o arcoseno do valor passado como argumento para o método. O valor retornado estará entre $-\pi/2$ e $\pi/2$. Se o argumento passado for `NaN`, ou se seu valor absoluto for maior que 1, o método retornará `NaN`. Se o argumento for zero positivo ou zero negativo, o argumento será retornado. Como exemplo, a expressão `Math.asin(1)` será aproximadamente igual a $\pi/2$.

*método* `acos` O método `acos` calcula e retorna o arco-coseno do valor passado como argumento para o método. O valor retornado estará entre 0 e $\pi$. Se o argumento passado for `NaN`, ou se seu valor absoluto for maior que 1, o método retornará `NaN`. Como exemplo, a expressão `Math.acos(1)` será aproximadamente igual a zero.

*método* `atan` O método `atan` calcula e retorna o arco-tangente do valor passado como argumento para o método. O valor retornado estará entre $-\pi/2$ e $\pi/2$. Se o argumento passado for `NaN`, zero positivo ou zero negativo, o argumento será retornado. Se o argumento for infinito positivo, o resultado será $\pi/2$, e se for infinito negativo, $-\pi/2$.

*método* `atan2` O método `atan` é comumente usado para a conversão de coordenadas retangulares para polares, para o cálculo do ângulo formado entre a origem do sistema de coordenadas e um ponto qualquer. Se as coordenadas forem dadas por $(x,y)$, o arcotangente de $y/x$ será o ângulo procurado, mas informação sobre a que quadrante o ângulo pertence seria perdida. O pacote `Math` provê o método `atan2`, que recebe dois argumentos ao invés de um, retornando um ângulo entre $-pi$ e $pi$ correspondente ao ângulo das coordenadas polares. Existem onze regras que ditam o resultado deste método, que cobrem os casos dos argumentos serem infinitos positivos, negativos ou `NaN`, que não serão listados aqui - basta saber que se qualquer um dos dois argumentos for `NaN`, o resultado será `NaN`, e se os dois argumentos forem infinitos, o resultado será um dos ângulos $\pi/4$, $3\pi/4$, $-3\pi/4$ ou $\pi/4$, dependendo dos sinais dos argumentos infinitos.

*método* `toDegrees` Dois últimos métodos trigonométricos da classe `Math` são métodos de conversão de unidades de ângulos: o método `toDegrees` recebe como argumento um valor do tipo `double`, representando um ângulo em radianos, e retorna este ângulo em graus. Desta forma, o resultado da expressão



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

*Math.toDegrees*(*Math.PI*/2) será 90. Similarmente, o método `toRadians` converterá o valor passado como argumento de graus para radianos, assim a expressão *Math.toRadians*(30.) será aproximadamente igual a $\pi/6$ ou 0.5235987755982988.

*método* `toRadians`

Para exemplificar alguns destes métodos consideremos a classe `Ponto2D`, já estudada, e reapresentada aqui com um método que permite a rotação de um ponto (instância da classe) em volta do ponto central. A figura 13.1 ilustra o processo de rotação de um ponto em duas dimensões.

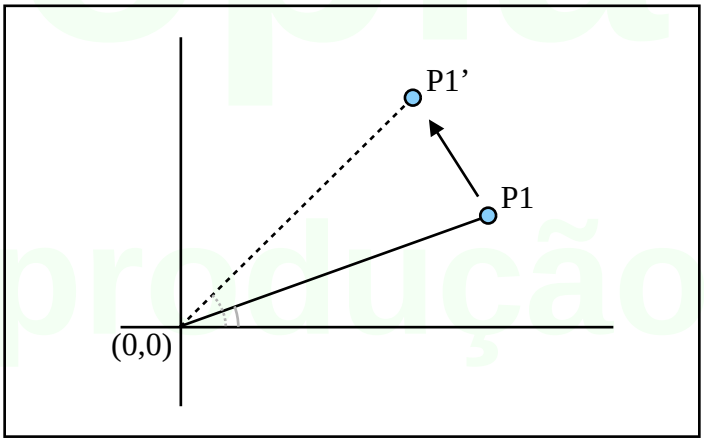



Figura 13.1: Rotação de um ponto em duas dimensões em volta da origem do sistema de coordenadas

A classe `Ponto2D` (listagem 13.1) contém um método que permite a rotação do ponto em torno da origem do sistema de coordenadas. O algortimo de rotação consiste no cálculo das coordenadas polares do ponto (sua distância da origem do sistema de coordenadas e ângulo em relação a esta origem), modificação do ângulo e na conversão de coordenadas polares modificadas para coordenadas retangulares.

Listagem 13.1: A classe `Ponto2D`, que encapsula um ponto no espaço cartesiano de duas dimensões (permitindo a rotação do ponto).

```
1  /**
2   * A classe Ponto2D, que encapsula um ponto no espaço cartesiano de duas dimensões.
3   * Esta implementação contém um método que permite a rotação do ponto.
4   */
5  class Ponto2D // declaração da classe
6    {
7   /**
8    * Declaração dos campos da classe
9    */
10   private double x,y; // as coordenadas do ponto no espaço bidimensional
11
12  /**
13   * O construtor para a classe Ponto2D, que recebe dois argumentos para inicializar os
14   * respectivos campos da classe. Não existem restrições aos valores que os dados podem
15   * receber, então a inicialização se limita a copiar os argumentos para os campos.
16   * @param _x o argumento correspondente à coordenada horizontal
17   * @param _y o argumento correspondente à coordenada vertical
18   */
19   Ponto2D(double _x,double _y)
20     {
21     x = _x;
22     y = _y;
23     } // fim do construtor
24
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
25  /**
26   * O método rotaciona recebe como argumento um ângulo e rotaciona o ponto usando
27   * este ângulo. O algoritmo transforma as coordenadas retangulares para polares,
28   * modifica o ângulo da coordenada polar e transforma novamente as coordenadas
29   * em retangulares. O argumento deve ser passado em graus, e será convertido de
30   * graus para radianos e vice-versa no método.
31   * @param ânguloDeRotação o ângulo para rotação do ponto, em graus
32   */
33   public void rotaciona(double ânguloDeRotação)
34     {
35     // Primeiro, determinamos o ângulo original e distância do centro
36     double distância = Math.sqrt(x*x+y*y);
37     double ânguloOriginal = Math.atan2(y,x);
38     // Convertemos este valor de radianos para graus
39     ânguloOriginal = Math.toDegrees(ânguloOriginal);
40     // Somamos o ângulo de rotação ao ângulo original
41     ânguloOriginal += ânguloDeRotação;
42     // Convertemos o ângulo para radianos para os cálculos subsequentes
43     ânguloOriginal = Math.toRadians(ânguloOriginal);
44     // Convertemos as coordenadas novamente para retangulares
45     x = distância*Math.cos(ânguloOriginal);
46     y = distância*Math.sin(ânguloOriginal);
47     }
48
49  /**
50   * O método getX retorna o valor da coordenada X deste ponto, já que o campo que a
51   * representa não pode ser acessado diretamente por ser privado.
52   * @return o valor da coordenada X
53   */
54   public double getX()
55     {
56     return x;
57     }
58
59  /**
60   * O método getY retorna o valor da coordenada Y deste ponto, já que o campo que a
61   * representa não pode ser acessado diretamente por ser privado.
62   * @return o valor da coordenada Y
63   */
64   public double getY()
65     {
66     return y;
67     }
68
69  /**
70   * O método toString é um método "mágico" que não recebe argumentos, e retorna uma
71   * String contendo os valores dos campos da classe formatados.
72   * @return uma String com os valores dos campos formatados como (x,y)
73   */
74   public String toString()
75     {
76     String resultado = "("+x+","+y+")";
77     return resultado;
78     } // fim do método toString
79
80   } // fim da classe Ponto2D
```

A classe `DemoPonto2D`, mostrada na listagem 13.2, demonstra o uso de uma instância da classe `Ponto2D`.

Listagem 13.2: A classe `DemoPonto2D`, que demonstra a criação de instância da classe `Ponto2D` e da rotação deste ponto.

```
1  /**
2   * A classe DemoPonto2D, que demonstra a criação de instância da classe Ponto2D
3   * e da rotação deste ponto.
4   */
```

```
5  class DemoPonto2D
6    {
7   /**
8    * O método main permite a execução desta classe. Este método cria uma instância da
9    * classe Ponto2D e executa várias vezes o método de rotação deste ponto.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     Ponto2D p = new Ponto2D(10,0); // coordenada inicial
16     System.out.println(p); // imprime coordenadas atuais
17     p.rotaciona(45); p.rotaciona(90); p.rotaciona(45); // rotaciona
18     System.out.println(p); // imprime coordenadas finais
19     } // fim do método main
20
21   } // fim da classe DemoPonto2D
```

### 13.3.3 Métodos exponenciais

A classe `Math` também contém alguns métodos para cálculo de funções exponenciais e logarítmicas. Estes métodos são listados nesta seção.

*método* `sqrt`

Um dos método mais usados em diversos tipos de cálculos é o método `sqrt`, que retorna a raiz quadrada do argumento passado. Se o argumento for `NaN` ou negativo, o valor `NaN` é retornado. Se o argumento for infinito positivo, zero negativo ou zero positivo o mesmo valor do argumento será retornado.

*método* `pow`

O método `pow` aceita dois argumentos do tipo `double` e retorna o valor do primeiro argumento elevado ao valor do segundo argumento (pi $x^y$, sendo que $x$ é o primeiro argumento e $y$ é o segundo argumento). Nada menos que dezessete regras regem o comportamento deste método, mas podemos sumarizar as mais importantes: para $y = 0$ e qualquer $x$, o método retornará 1.0; para $y = 1$ e qualquer $x$ o método retornará $x$; se $x$ e/ou $y$ forem `NaN` e $y$ for diferente de 0, o resultado será `NaN`, e se $x < 0$ e $y$ não for um valor inteiro, o resultado será `NaN`.

*método* `log`

O método `log` aceita um argumento e retorna o valor do logaritmo (base $e$ ou natural) do argumento. Se o argumento for `NaN` ou negativo, o método retornará `NaN`. Se o argumento for infinito positivo, o método retornará infinito positivo. Se o argumento for zero (positivo ou negativo) o método retornará infinito negativo.

*método* `exp`

O método `exp` recebe um argumento e retorna o valor da constante $e$ elevado ao valor do argumento. Se o argumento for `NaN` o resultado será `NaN`. Se o argumento for infinito positivo, o resultado será infinito positivo. Se o argumento for infinito negativo, o resultado será zero positivo.

Para ilustrar o uso de algumas destes métodos, consideremos o problema do cálculo de uma integral definida entre dois pontos. A regra do trapézio divide a área da curva da função da qual queremos calcular a integral em $N$ intervalos, considerando cada um com um formato de um trapézio, e calcula a somatória das áreas de todos estes trapézios, como ilustrado na figura 13.2.

A figura 13.2 mostra como o valor da integral de uma função qualquer entre os valores $a$ e $b$ pode ser calculada. Para isto, precisamos dividir o intervalo $b-a$ em vários subintervalos e calcular a área de cada trapézio como sendo $f(x_i) + f(x_{i+1})/2$, onde $f()$ é a função da qual queremos calcular a integral, $x_i$ é o valor no início de um subintervalo e $x_{i+1}$ é o valor no fim de um subintervalo.

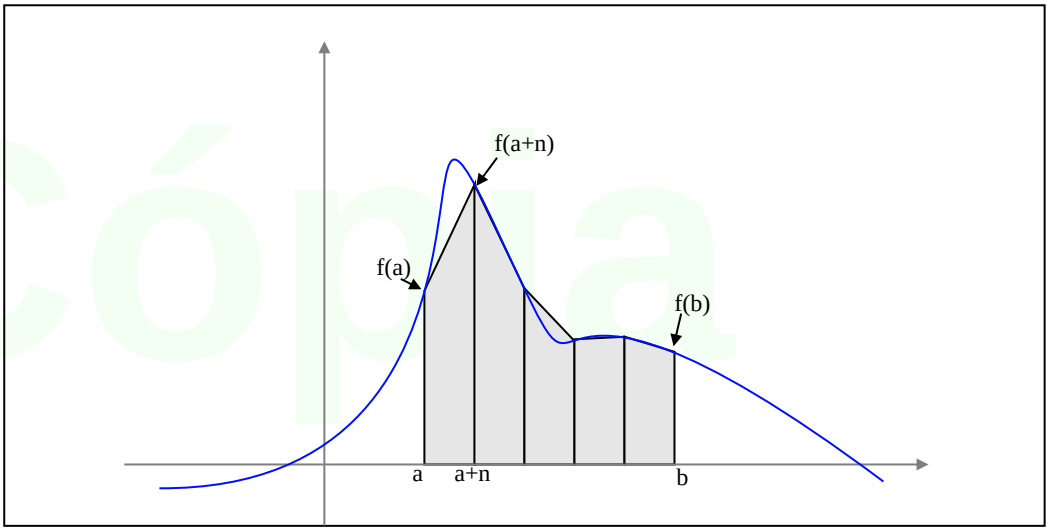



Figura 13.2: Representação gráfica da aproximação da integral pela somatória das áreas de trapézios.

A classe `RegraDoTrapezio` demonstra uma maneira de implementar a solução deste algoritmo para a função $2/e^x$. A classe contém um método `main` que define alguns dos parâmetros do algoritmo (valores para $a$ e $b$, número de subintervalos) e executa os cálculos chamando uma função (método estático) que retorna $2/e^x$ onde $x$ é o argumento passado. A classe `RegraDoTrapezio` é mostrada na listagem 13.3.

Listagem 13.3: A classe `RegraDoTrapezio`, que calcula o valor da integral definida de uma função usando a regra do trapézio.

```
1  /**
2   * A classe RegraDoTrapezio, que calcula o valor da integral definida de uma função
3   * usando a regra do trapézio. Esta classe assume que a função que representa a
4   * integral está codificada como um método estático.
5   */
6  class RegraDoTrapezio // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método calcula diretamente
10   * o valor da integral, usando como subrotina um método estático que calcula o valor
11   * da função sendo considerada.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     // Declaramos e definimos o número de subintervalos
18     int totalSubIntervalos = 100;
19     // Declaramos e definimos o intervalo inferior e superior da integral
20     double intervaloInferior = 2.0;
21     double intervaloSuperior = 4.0;
22     // Declaramos e calculamos delta, a largura do trapézio entre os subintervalos
23     double delta = (intervaloSuperior-intervaloInferior)/totalSubIntervalos;
24     // Preparamos para calcular a somatória, inicializando esta com zero e
25     // inicializando o primeiro ponto do trapézio com o intervalo inferior
26     double somatória = 0;
27     double xi = intervaloInferior;
28     // Um laço calculará a somatória. Este laço deve ser executado totalSubIntervalos
29     // vezes, e para cada vez o termo da somatória deve ser calculado e acumulado
30     for(int subIntervalo=0;subIntervalo<totalSubIntervalos;subIntervalo++)
```

```
31        {
32        double termo = 0.5*(função(xi)+função(xi+delta))*delta; // o termo da somatória
33        somatória += termo; // é acumulado nela
34        xi += delta; // o próximo intervalo é calculado (somente um lado do trapézio)
35        }
36      // O resultado é impresso
37      System.out.println("O valor da integral é "+somatória);
38      } // fim do método main
39
40    /**
41     * O método função calcula o valor da função sendo considerada para um determinado
42     * valor. A função representada neste método é 2/(e^x).
43     * @param x o valor para cálculo da função
44     * @return o valor da função para x
45     */
46    private static double função(double x)
47      {
48      return 2.0/Math.exp(x);
49      }
50
51    } // fim da classe RegraDoTrapezio
```

Outro exemplo de uso dos métodos da classe `Math` para cálculos matemáticos é dado pela classe `RaizDeFuncao`, que usa o método da bisecção sucessiva para achar a raiz (ponto onde a função vale zero) de uma função. O método da bisecção sucessiva tenta localizar uma raiz da função partindo de um intervalo inicial. O algoritmo do método é descrito a seguir:

1. Um intervalo inicial é selecionado de forma que o valor da função nos pontos extremos do intervalo tenha sinais opostos.
2. Localizamos o ponto central do intervalo (ponto médio entre os extremos do intervalo). Se o valor da função no ponto médio for zero, consideramos o ponto médio como zero da função e interrompemos a procura.
3. Se o tamanho do intervalo de busca for muito pequeno (menor que uma dada constante) consideramos que o ponto médio do intervalo é uma aproximação boa o suficiente do zero da função, e interrompemos o algoritmo.
4. Se o sinal da função no ponto médio for igual ao sinal da função no limite inferior de busca, consideramos que o zero da função deve estar entre o ponto médio e o limite superior de busca. Fazemos com que o novo limite inferior de busca seja o ponto médio e voltamos ao passo 2.
5. Se o sinal da função no ponto médio for igual ao sinal da função no limite superior de busca, consideramos que o zero da função deve estar entre o limite inferior de busca e o ponto médio. Fazemos com que o novo limite superior de busca seja o ponto médio e voltamos ao passo 2.

Uma ilustração deste algoritmo é mostrada na figura 13.3. Nesta figura, inicialmente procuramos a raiz no intervalo delimitado pelos pontos $i_0$ e $i_1$. Consideramos $i_2$ como sendo o ponto central deste intervalo, e observamos que $f(i_2)$ e $f(i_1)$ tem o mesmo sinal, assim modificamos o intervalo de busca para $i_0$ e $i_2$, calculando o novo ponto central como $i_3$. Podemos ver que com mais algumas iterações deste algoritmo chegamos cada vez mais perto da raiz da função.

A classe `RaizDeFuncao`, mostrada na listagem 13.4, demonstra uma implementação do algoritmo de bisecção sucessiva.

Listagem 13.4: A classe `RaizDeFuncao`, que calcula o zero de uma função qualquer usando o método de bisecção sucessiva.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

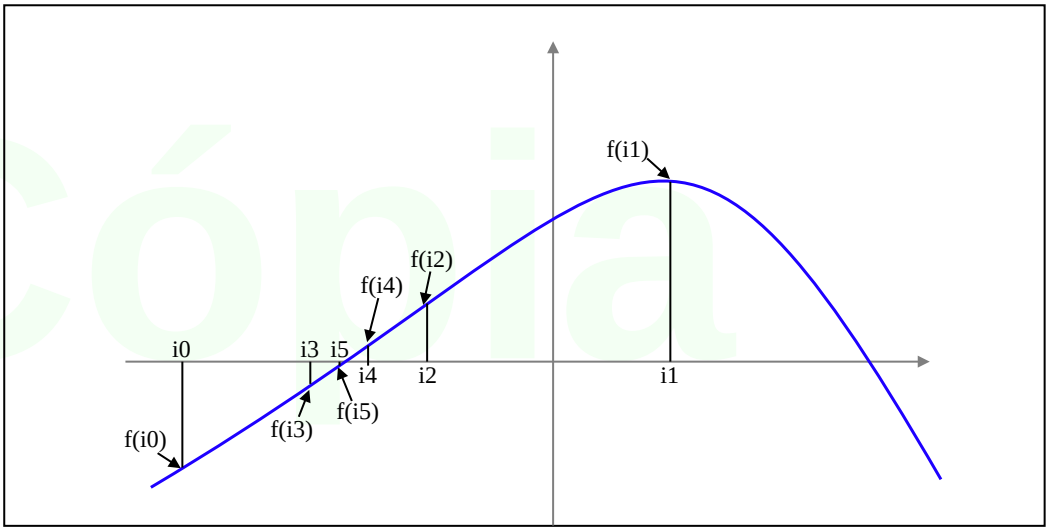



Figura 13.3: Ilustração do algoritmo de bisecção sucessiva para localização de raízes de uma função.

```
1  /**
2   * A classe RaizDeFuncao, que calcula o zero de uma função qualquer usando o método
3   * de bisecção sucessiva. Esta classe assume que a função para a qual queremos calcular
4   * o zero está codificada como um método estático.
5   */
6  class RaizDeFuncao // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método calcula diretamente
10   * o zero da função, usando como subrotina um método estático que calcula o valor
11   * da função sendo considerada.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     // Declaramos e inicializamos o intervalo inicial de procura da raiz.
18     double limiteInferior = -2.0;
19     double limiteSuperior =  2.0;
20     // Declaramos uma variável para conter a raiz da função.
21     double raiz;
22     // Declaramos a precisão desejada no cálculo.
23     double intervaloMínimo = 0.00001;
24     // Executamos o algoritmo "para sempre" (até que uma condição de parada ocorra)
25     while(true)
26       {
27       // Calculamos o ponto do meio do intervalo.
28       double pontoCentral = limiteInferior+(limiteSuperior-limiteInferior)/2.0;
29       // Se o valor da função no ponto central for exatamente zero, consideramos que
30       // o ponto central é a raiz e saímos do laço.
31       if (função(pontoCentral) == 0.)
32         {
33         raiz = pontoCentral;
34         break;
35         }
36       // Se o intervalo for menor que o intervalo mínimo, consideramos que o ponto
37       // central é uma aproximação boa o suficiente da raiz e saímos do laço.
38       if (limiteSuperior-limiteInferior < intervaloMínimo)
39         {
40         raiz = pontoCentral;
41         break;
42         }
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
43        // Se o sinal da função no ponto central for igual a do limite inferior, a raiz
44        // estará entre o ponto central e o limite superior. Modificamos então o limite
45        // inferior e continuamos a busca.
46        if (sinal(função(pontoCentral)) == sinal(função(limiteInferior)))
47          limiteInferior = pontoCentral;
48        // Se o sinal da função no ponto central for igual a do limite superior, a raiz
49        // estará entre o limite inferior e o ponto central. Modificamos então o limite
50        // superior e continuamos a busca.
51        if (sinal(função(pontoCentral)) == sinal(função(limiteSuperior)))
52          limiteSuperior = pontoCentral;
53        }
54      // Ao fim do laço, imprimimos o resultado.
55      System.out.println("A raiz é "+raiz);
56      } // fim do método main
57
58   /**
59    * O método sinal calcula o sinal do valor passado como argumento e retorna -1 para
60    * sinais negativos, +1 para sinais positivos e zero se o valor for exatamente zero.
61    * @param x o valor para cálculo do sinal
62    * @return o sinal de x (-1, 0 ou +1)
63    */
64    private static int sinal(double x)
65      {
66      if (x == 0.) return 0;
67      else if (x < 0) return -1;
68      else return +1;
69      }
70
71   /**
72    * O método função calcula o valor da função sendo considerada para um determinado
73    * valor. A função representada neste método é sen(x)+cos(x)
74    * @param x o valor para cálculo da função
75    * @return o valor da função para x
76    */
77    private static double função(double x)
78      {
79      return Math.sin(x)+Math.cos(x);
80      }
81
82    } // fim da classe RaizDeFuncao
```

### 13.3.4 Métodos para cálculo de valores aleatórios

*método* `random`

A classe `Math` provê um método muito útil para uso em jogos e simulações: o método `random`, que não recebe argumentos e retorna um valor pseudo-aleatório, do tipo `double`, maior ou igual a zero e menor do que um, que será aparentemente diferente cada vez que o método for executado. Este valor pode ser usado para tomada de decisões que podem ser diferentes a cada execução de um método ou aplicação, adicionando variações a estes. O valor retornado é dito pseudo-aleatório porque o mecanismo de geração dos valores é conhecido, portanto o próximo valor a ser retornado pelo método pode ser previsto. Em uma sequência de números realmente aleatórios, não deveria ser possível a previsão do próximo valor.

*classe* `Random`

Números inteiros pseudo-aleatórios maiores ou iguais a $a$ e menores que $b$ podem ser obtidos usando-se o trecho de código `(int)(a+(Math.random()*(b-a)))`. Valores booleanos pseudo-aleatórios podem ser obtidos usando-se `(Math.random() > 0.5)`. Aplicações que usam muito o cálculo de números aleatórios podem implementar estes trechos de código como rotinas internas da classe. A classe `Random`, do pacote `java.util`, provê outros métodos de geração de valores pseudo-aleatórios.

Como demonstração do método `random` da classe `Math`, consideremos o *algoritmo de Buffon*, um método bastante curioso para o cálculo aproximado de $\pi$. Este algoritmo consiste em lançar mui-

tas agulhas de comprimento constante sobre uma folha onde estão desenhadas linhas paralelas, separadas por duas vezes o comprimento das agulhas. De acordo com o algoritmo, o resultado da divisão do número de agulhas lançadas pelo número de agulhas que tocam as linhas paralelas deverá ser aproximadamente *pi*. A figura 13.4 ilustra este algoritmo, chamando de *N* o comprimento da agulha.

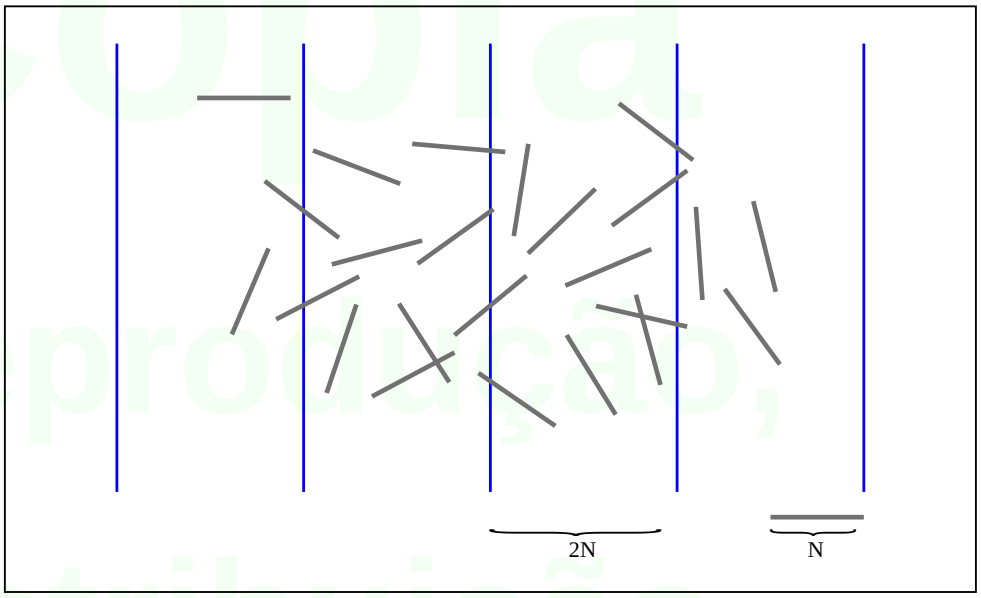



Figura 13.4: Representação gráfica do algoritmo de Buffon.

Para ilustrar o algoritmo de Buffon, vamos usar algumas classes que representam seus elementos. A primeira classe será a classe `Agulha`, que representa uma agulha de comprimento constante (0.5, neste exemplo). Esta classe terá um método que funcionará como uma fábrica de instâncias da classe `Agulha` que criará agulhas usando um ponto de origem e ângulo de inclinação pseudo-aleatórios. A origem de cada agulha estará entre as coordenadas $(-1,-1)$ e $(+1,+1)$ e o ângulo entre 0 e 360 graus. A classe `Agulha` é mostrada na listagem 13.5.

Listagem 13.5: A classe `Agulha`, que representa uma agulha de comprimento fixo.

```
1  /**
2   * A classe Agulha, que representa uma agulha de comprimento fixo (uma linha em duas
3   * dimensões). Instâncias desta classe serão usadas para demonstrar o algoritmo de
4   * Buffon.
5   */
6  class Agulha // declaração da classe
7    {
8   /**
9    * Declaração dos campos da classe
10   */
11   private Ponto2D p1,p2; // os pontos extremos da linha no espaço bidimensional
12   private double ângulo; // o ângulo (em graus) de inclinação desta agulha
13   // Este campo, representando o tamanho padrão da agulha, é constante
14   public static final double tamanho = 0.5;
15
16  /**
17   * O construtor da classe, que recebe como argumentos um dos pontos extremos da
18   * agulha e um valor do tipo double, correspondente ao ângulo (em graus) de
19   * inclinação da agulha.
20   * @param origem a origem da agulha (um dos seus pontos extremos)
21   * @param a o ângulo desta agulha com a origem
22   */
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
23   Agulha(Ponto2D origem,double a)
24     {
25     p1 = origem; // inicializa o primeiro ponto com o argumento
26     ângulo = a; // inicializa o ângulo com o argumento
27     // Calculamos o segundo ponto com o ângulo e o tamanho constante. Primeiro
28     // calculamos o valor do ângulo em radianos
29     double ânguloEmRadianos = Math.toRadians(ângulo);
30     // Calculamos as coordenadas X e Y da outra extremidade da agulha
31     double novoX = p1.getX()+tamanho*Math.cos(ânguloEmRadianos);
32     double novoY = p1.getY()+tamanho*Math.sin(ânguloEmRadianos);
33     // Criamos a nova instância
34     p2 = new Ponto2D(novoX,novoY);
35     } // fim do construtor
36
37   /**
38    * O método criaAleatória retorna uma nova instância da classe Agulha, criada com
39    * o primeiro ponto localizado em um espaço aleatório entre as coordenadas (-1,1) e
40    * (1,1) e o ângulo aleatório entre 0 e 360 graus. Este método é uma fábrica de
41    * instâncias, podendo ser usado para a criação rápida de instâncias desta classe.
42    * @return uma nova instância da classe Agulha em posição aleatória.
43    */
44   public static Agulha criaAleatória()
45     {
46     // Calculamos as coordenadas do primeiro ponto
47     double primeiroX = Math.random()*2.0-1.0;
48     double primeiroY = Math.random()*2.0-1.0;
49     double ângulo = Math.random()*360.0;
50     // Criamos uma instância desta classe com os valores calculados
51     Agulha temporário = new Agulha(new Ponto2D(primeiroX,primeiroY),
52                                    ângulo);
53     return temporário; // retornamos a referência inicializada
54     } // fim do método criaAleatória
55
56   /**
57    * O método getP1 retorna as coordenadas do primeiro ponto da agulha como uma
58    * instância da classe Ponto2D.
59    * @return as coordenadas do primeiro ponto da agulha
60    */
61   public Ponto2D getP1()
62     {
63     return p1;
64     }
65
66   /**
67    * O método getP2 retorna as coordenadas do segundo ponto da agulha como uma
68    * instância da classe Ponto2D.
69    * @return as coordenadas do segundo ponto da agulha
70    */
71   public Ponto2D getP2()
72     {
73     return p2;
74     }
75
76   /**
77    * O método toString retorna os campos desta agulha, formatados em uma String.
78    * @return uma String com os valores dos campos formatados
79    */
80   public String toString()
81     {
82     String resultado = p1+"-"+p2+", a="+ângulo;
83     return resultado;
84     } // fim do método toString
85
86   } // fim da classe Agulha
```

As agulhas do algoritmo de Buffon devem ser lançadas sobre uma mesa ou papel com linhas paralelas espaçadas entre si. Para calcular a proporção de agulhas que atravessam estas linhas, métodos que comparam coordenadas de agulhas e linhas devem ser escritos. As linhas serão

representadas por instâncias da classe `LinhaVerticalInfinita` - as linhas não precisam necessariamente ser infinitas, mas as coordenadas verticais das linhas não serão usadas, e a comparação de intersecção fica mais simples assim. A classe `LinhaVerticalInfinita` é mostrada na listagem 13.6.

Listagem 13.6: A classe `LinhaVerticalInfinita`, que representa uma linha vertical infinita.

```
/**
 * A classe LinhaVerticalInfinita, que representa uma linha vertical infinita.
 * Instâncias desta classe serão usadas para demonstrar o algoritmo de Buffon.
 */
class LinhaVerticalInfinita // declaração da classe
  {
 /**
  * Declaração dos campos da classe
  */
  private double x; // a coordenada X (horizontal) desta linha

 /**
  * O construtor da classe, que recebe como argumento a coordenada horizontal da linha.
  * @param x a coordenada horizontal da linha
  */
  LinhaVerticalInfinita(double x)
    {
    this.x = x; // inicializamos o campo
    } // fim do construtor

 /**
  * O método intercepta recebe como argumento uma instância da classe Agulha e
  * retorna true se esta instância de Agulha intercepta a instância da linha
  * vertical. O algoritmo de intercepção é bem simples: se um dos pontos da agulha
  * está em um lado da linha e o outro ponto da agulha no outro lado, a agulha
  * intercepta a linha. Se qualquer um dos dois pontos está sobre a linha, também
  * consideramos que a agulha intercepta a linha. Somente as coordenadas x serão
  * consideradas.
  * @param a uma instância da classe Agulha
  * @return true se a agulha intercepta a linha vertical infinita
  */
  public boolean intercepta(Agulha a)
    {
    // Primeiro caso: o primeiro ponto está à esquerda da linha e o segundo à direita.
    if ((a.getP1().getX() < x) && (x < a.getP2().getX())) return true;
    // Segundo caso: o primeiro ponto está à direita da linha e o segundo à esquerda.
    if ((a.getP1().getX() > x) && (x > a.getP2().getX())) return true;
    // Terceiro caso: um dos pontos da agulha está sobre a linha vertical
    if ((a.getP1().getX() == x) || (a.getP2().getX() == x)) return true;
    // Se nenhum dos casos ocorreu, a agulha não intercepta a linha
    return false;
    } // fim do método intercepta

 /**
  * O método toString retorna os campos desta linha vertical, formatados em uma String.
  * @return uma String com os valores dos campos formatados
  */
  public String toString()
    {
    String resultado = "Linha vertical na coordenada "+x;
    return resultado;
    } // fim do método toString

  } // fim da classe LinhaVerticalInfinita
```

Finalmente, escrevemos uma aplicação que usa instâncias das classes `LinhaVerticalInfinita` e `Agulha` para demonstrar o algoritmo de Buffon. O método `main` da classe `Buffon` declara um array com três instâncias da classe `LinhaVerticalInfinita` e pede ao usuário que entre

um número de agulhas a ser lançado, criando o número correspondente de agulhas e verificando quantas cruzam as linhas verticais. A classe `Buffon` é mostrada na listagem 13.7.

Listagem 13.7: A classe `Buffon`, que simula o algoritmo de Buffon para cálculo do valor de pi.

```
/**
 * A classe Buffon, que simula o algoritmo de Buffon para cálculo do valor de pi.
 */
class Buffon // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método cria duas linhas
  * verticais infinitas e muitas instâncias de agulhas, e verifica quantas agulhas
  * atravessam qualquer uma das linhas verticais. A razão entre o número de agulhas
  * criadas e o número de agulhas que atravessam as linhas verticais deve convergir
  * para pi.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Declaramos algumas variáveis para acumular o total de agulhas e o número de
    // agulhas que interceptam as linhas. O número total de agulhas a ser lançado
    // será lido do teclado.
    int totalDeAgulhas,agulhasInterceptadas=0;
    System.out.print("Número total de agulhas:");
    totalDeAgulhas = Keyboard.readInt();
    // Declaramos algumas instâncias da classe LinhaVerticalInfinita como um array
    LinhaVerticalInfinita[] linhas = new LinhaVerticalInfinita[3];
    // Inicializamos as instâncias da classe LinhaVerticalInfinita, usando duas vezes
    // os comprimentos das agulhas para espaçá-las.
    linhas[0] = new LinhaVerticalInfinita(-2*Agulha.tamanho);
    linhas[1] = new LinhaVerticalInfinita( 0);
    linhas[2] = new LinhaVerticalInfinita(+2*Agulha.tamanho);
    // Fazemos a simulação com as agulhas
    for(int agulhas=0;agulhas<totalDeAgulhas;agulhas++)
      {
      // Criamos uma instância usando a fábrica na classe Agulha
      Agulha estaAgulha = Agulha.criaAleatória();
      // Verificamos se a agulha criada intercepta alguma linha do array
      for(int qualLinha=0;qualLinha<linhas.length;qualLinha++)
        {
        if (linhas[qualLinha].intercepta(estaAgulha))
          agulhasInterceptadas++;
        }
      }
    // Imprimimos o resultado final, convertendo a divisão para ponto flutuante
    System.out.println(totalDeAgulhas+"/"+agulhasInterceptadas+"="+
                       (1.0*totalDeAgulhas/agulhasInterceptadas));
    } // fim do método main

  } // fim da classe Buffon
```

Uma execução da aplicação com cem milhões de agulhas obteve o valor 3.141475764158482 para *pi*.

## 13.4 Representando valores com precisão arbitrária

Valores dos tipos nativos de Java tem limitações claras em relação aos números que podem ser representados. Eventualmente seria interessante ou útil poder extrapolar estes limites, para permitir cálculos com resultados mais precisos.

O pacote `java.math` (que não deve ser confundido com a classe `Math` no pacote `java.lang`) contém duas classes que encapsulam valores com precisão muito maior do que as encontradas em tipos nativos. Estas classes representam valores com *precisão arbitrária*, com operações que não sofrem perda de precisão[3]. Estas classes são apresentadas nas seções seguintes.

### 13.4.1 A classe `BigInteger`

*classe* `BigInteger`

A classe `BigInteger` encapsula valores inteiros com precisão arbitrária, e contém vários métodos para processar estes valores. Como instâncias de uma classe, números inteiros de precisão arbitrária não poderão ser manipulados diretamente por operadores como `+`, `*`, `==`, etc. - as operações com estes números serão sempre implementadas através de métodos. É importante notar que instâncias da classe `BigInteger` são imutáveis - uma vez criadas, não podem ser modificadas, e os métodos que implementam operações retornam sempre novas instâncias da classe. Nesta seção veremos os métodos mais úteis da classe e um exemplo mais completo de seu uso.

*campo* `ZERO`

*campo* `ONE`

A classe `BigInteger` contém dois campos estáticos públicos que representam os valores mais utilizados como instâncias da classe: o campo `ZERO` representa uma instância da classe que encapsula o valor zero e o campo `ONE` encapsula o valor um. Instâncias da classe que representem estes valores podem ser construídas diretamente como, por exemplo, `BigInteger soma = BigInteger.ZERO;`.

*construtores para a classe* `BigInteger`

Existem outras maneiras de construir instâncias da classe `BigInteger`. A classe provê quatro construtores interessantes: o primeiro aceita como argumento uma instância da classe `String` que deve representar um valor numérico precedido opcionalmente de um caracter `'-'`. Os caracteres da string serão analisados para a criação da instância da classe, e caso algum caracter inválido apareça, a exceção `NumberFormatException` ocorrerá. Caracteres inválidos são todos os caracteres menos os dígitos e o caracter `'-'`, e este somente pode aparecer na primeira posição.

Um construtor similar recebe, além da string, um valor inteiro entre 2 e 36 que será considerado a base do valor a ser encapsulado - assim será possível criar instâncias com bases que não sejam a decimal. Como exemplos, `new BigInteger("10000000000000000000");` retornará uma instância da classe contendo o valor cem quintilhões, e `new BigInteger("-FFFFFFFFFFFFFFFF", 16);` retornará uma instância da classe que encapsula o valor $-18446744073709551615$, ambos fora da faixa de valores que o tipo nativo `long` pode representar.

*método* `valueOf`

Um outro método da classe `BigInteger` permite a criação de instâncias da classe: o método `valueOf` recebe como argumento um valor do tipo `long` e retorna uma nova instância da classe `BigInteger` que encapsula o valor passado como argumento. O método é estático, então não é necessária a criação de uma instância para que possamos executá-lo.

*método* `compareTo`

Uma instância da classe `BigInteger` pode ser comparada com outra de duas formas: o método `compareTo` recebe como argumento uma instância da própria classe e retorna $-1$ se o valor encapsulado for menor que o passado como argumento, 0 se forem iguais e $+1$ se o valor encapsulado for maior que o passado como argumento. Este método pode ser usado para emular as seis operações básicas de comparação (`==`, `!=`, `>=`, `>`, `<=` e `<`) - basta comparar o resultado deste método com o valor zero, usando um destes operadores. Como exemplo, se as instâncias `n1` e `n2` da classe `BigInteger` forem criadas como `n1 = new BigInteger("5000000000000000000");` e `n2`

[3] Na verdade, como estes valores são armazenados internamente como arrays, que tem um número máximo de posições limitado, podemos considerar que haverá um limite de precisão, mas muito além das aplicações reais mais exigentes.

`= new BigInteger("5000000000000000001");` a operação `(n1.compareTo(n2) <= 0)` retornará `true` pois `n1` é menor ou igual a `n2`, e a operação `(n2.compareTo(n1) == 0)` retornará `false` pois `n2` é diferente de `n1`.

O método compara o valor encapsulado na instância que executar o método com o valor encapsulado na instância passada como argumento, retornando `true` somente se as duas representarem o mesmo valor.

*método* `equals`

Existem outros dois métodos de comparação de instâncias de `BigInteger`: o método `max` recebe um argumento e retorna o maior dos valores (comparando o da instância que executa o método com o passado como argumento). Similarmente, o método `min` retorna o menor dos valores, comparando o encapsulado pela instância que executa o método com o passado como argumento.

*método* `max`

*método* `min`

As quatro operações básicas podem ser executadas usando uma instância da classe `BigInteger` representando o primeiro operador, que executará o método correspondente que por sua vez receberá o segundo operador como argumento.

A primeira operação básica é adição, efetuada pelo método `add`, que retornará uma nova instância da classe encapsulando o resultado da adição. Como exemplo, consideremos o trecho de código `BigInteger s1,s2; s1 = new BigInteger("9"); s2 = s1.add(BigInteger.ONE);`, que resultará na instância `s2` encapsulando o valor 10. Métodos de operações básicas podem ser chamados em cascata, como no exemplo `BigInteger s1 = new BigInteger("1").add(new BigInteger("3")).add(new BigInteger("5")).add(new BigInteger("7"));`, que resultará em `s1` encapsulando o valor 16. Note que os argumentos para o método `add` devem ser instâncias da classe `BigInteger`, que são construídos sem referências, para uso pelos métodos.

*método* `add`

Outra operação básica é a de subtração, efetuada pelo método `subtract` e que retornará uma nova instância da classe contendo o resultado da subtração dos valores encapsulados pela instância que executar o método e da que for passada como argumento. Como exemplo, o trecho de código `BigInteger b1 = new BigInteger("100000"); BigInteger b2 = b1.subtract(new BigInteger("2"));` fará com que a instância `b2` encapsule o valor 99998.

*método* `subtract`

A multiplicação de duas instâncias da classe `BigInteger` é feita pelo método `multiply`, que também recebe um argumento e retorna uma nova instância da mesma classe. Um exemplo de uso deste método pode ser dado pelo trecho de código `BigInteger m = new BigInteger("999"); System.out.println(m.multiply(m));` que imprimirá 998001.

*método* `multiply`

A classe `BigInteger` também tem métodos para a divisãode valores encapsulados por valores passados como argumentos. O método `divide` faz esta operação, retornando somente a parte inteira da divisão, uma vez que instâncias de `BigInteger` não armazenam casas decimais. Desta forma, se o trecho de código `BigInteger d = new BigInteger("1000"); BigInteger r = d.divide(new BigInteger("3"));` for executado, a instância `r` estará encapsulando o valor 333. Se o valor do argumento passado para o método `divide` for igual a zero, a exceção `ArithmeticException` ocorrerá.

*método* `divide`

O resto da divisão de uma instância de `BigInteger` por outra pode ser obtido com o método `remainder`, que é funcionamente igual ao operador `%`. Assim, ao final do trecho de código `BigInteger d = new BigInteger("12345678"); BigInteger r = d.remainder(new BigInteger("7"));` a instância `r` estará armazenando o valor 2. Se o valor do argumento passado para o método `divide` for igual a zero, a exceção `ArithmeticException` ocorrerá. O método

*método* `remainder`

*método* `mod`



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

`mod` funciona de modo similar ao método `remainder`, exceto que somente instâncias contendo valores positivos são retornadas.

*método* `pow` É possível efetuarmos a exponenciação de um valor `BigInteger` por um valor inteiro com o método `pow`. Este método recebe como argumento um valor inteiro (e não uma instância da própria classe `BigInteger`, como os outros métodos da classe) e retorna o valor original encapsulado elevado ao valor passado como argumento. O valor passado como argumento não pode ser negativo, caso contrário uma exceção `ArithmeticException` ocorrerá. Como exemplo, o trecho de código `BigInteger d = new BigInteger("2"); System.out.println(d.pow(64));` imprimirá 18446744073709551616.

*método* `negate` O método `negate` (que não recebe argumentos) retorna uma nova instância da classe que é o valor negativo do valor encapsulado, sendo equivalente à multiplicação da instância por `new BigInteger("-1")`. Um outro método que manipula sinais de instâncias de `BigInteger` é o

*método* `abs` método `abs` (também sem argumentos), que retorna uma nova instância da classe contendo o valor encapsulado mas sem o sinal (sendo sempre positivo). Um outro método aparentado é o

*método* `signum` `signum`, também sem argumentos, que retorna o sinal da instância de `BigInteger` como um valor inteiro que será igual a $-1$, 0 ou $+1$ se o valor encapsulado for negativo, zero ou positivo, respectivamente.

*método* `intValue` A classe fornece vários métodos para retornar o valor encapsulado como um valor de tipo nativo: o método `intValue` retorna um valor inteiro correspondente aos 32 bits menos significativos da instância de `BigInteger`, o que possivelmente resultará em valores aparentemente incoerentes se o valor encapsulado no `BigInteger` for muito grande para ser contido por um `int`. Por exemplo, a execução da expressão `BigInteger d = new BigInteger("2").pow(64); System.out.println(d.intValue());` imprimirá zero.

*método* `longValue` O método `longValue` efetua a mesma conversão, mas usando os 64 bits menos significativos. Desta forma, ao final da expressão `BigInteger d = new BigInteger("2").pow(32); long w = d.longValue();` o valor `w` conterá 4294967296.

*método* `floatValue` O valor encapsulado também pode ser retornado em um valor nativo de ponto flutuante com o método `floatValue`. Caso o valor representado pelo `BigInteger` seja muito grande para ser representado por um valor do tipo `float`, este será considerado como infinito, negativo ou positivo. Assim, se o resultado da expressão `new BigInteger("2").pow(1000);` (um número com 302 dígitos) for convertido para um valor do tipo `float`, o resultado será `Infinity`.

*método* `doubleValue` O método `doubleValue` funciona da mesma maneira, mas convertendo o valor para um valor do tipo `double`. Se o resultado de `new BigInteger("2").pow(1000);` fosse convertido com o método `doubleValue`, o resultado seria $1.0715086071862673E301$ - houve perda de precisão, mas o valor é aproximadamente representável por um `double`.

*método* `toString` Épossível converter os valores encapsulados por instâncias da classe `BigInteger` para strings, sem perda de precisão (mas evidentemente sem a capacidade de processamento como tipos nativos), usando duas versões do método `toString`: uma é a tradicional, que não recebe argumentos e retorna uma string onde cada caracter corresponderá a um dígito do `BigInteger`. Uma outra forma do método `toString` aceita um valor inteiro como argumento, que será usado como a base que será usada para a conversão. Assim, a execução do trecho de código `BigInteger b = new BigInteger("2").pow(1000); String s = b.toString(16);` armazenará na string `s` uma string composta do caracter `'1'` seguido de 250 caracteres `'0'`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Como exemplo do uso de alguns métodos da classe `BigInteger`, consideremos o problema do cálculo do fatorial de um número. Na classe `ProbabilidadeBasica`, mostrada na listagem 7.7, fizemos o cálculo do fatorial usando um valor do tipo `double`, pois se usássemos um valor do tipo `long`, somente poderíamos calcular o fatorial de, no máximo, 20. Usando instâncias da classe `BigInteger` para armazenar o valor do fatorial, podemos calculá-lo com precisão arbitrária. A classe `BigFatorial`, mostrada na listagem 13.8, demonstra como isto pode ser feito.

Listagem 13.8: A classe `BigFatorial`, que demonstra o uso da classe `BigInteger` para o cálculo de fatoriais.

```
/* Precisaremos da classe BigInteger que está no pacote java.math, importaremos
   todas as classes deste pacote. */
import java.math.*;

/**
 * A classe BigFatorial contém um método estático que calcula o fatorial de um
 * número usando uma instância da classe BigInteger para armazenar os resultados.
 * Desta forma fatoriais de valores relativamente elevados poderão ser calculados
 * sem erros de overflow.
 */
class BigFatorial // declaração da classe
  {
 /**
  * O método fatorial calcula o fatorial de um valor inteiro, retornando o resultado
  * como uma instância da classe BigInteger (para evitar overflow).
  * @param valor o valor para o qual calcularemos o fatorial
  * @return o fatorial do valor passado como argumento (instância da classe BigInteger)
  */
  public static BigInteger fatorial(int valor)
    {
    // O acumulador do fatorial deve ser uma instância da classe BigInteger. Usamos
    // o campo constante que vale um para inicializar o acumulador.
    BigInteger fatorial = BigInteger.ONE; // inicializamos com 1
    // Executamos um laço que fará a iteração entre 1 e o valor cujo fatorial estamos
    // calculando.
    for(int contador=1;contador<=valor;contador++)
      {
      // Criamos uma outra instância da classe BigInteger contendo o valor a ser
      // multiplicado, usando a fábrica de instâncias valueOf.
      BigInteger multiplicando = BigInteger.valueOf(contador);
      // Multiplica o valor do fatorial pelo multiplicando, usando um método específico
      // da classe BigInteger.
      fatorial = fatorial.multiply(multiplicando);
      }
    // Retorna como resultado do método o valor do fatorial.
    return fatorial;
    }

 /**
  * O método main permite a execução desta classe. Este método chama o método
  * fatorial com vários valores como argumentos.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    // Calculamos o fatorial de 20...
    BigInteger fat = fatorial(20);
    System.out.println(fat);
    // ... o fatorial de 100...
    System.out.println(fatorial(100));
    // ... e o fatorial de 50000.
    System.out.println(fatorial(50000));
    }
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
55
56  } // fim da classe BigFatorial
```

Podemos notar na listagem 13.8 que todos os cálculos do fatorial devem ser feitos usando instâncias da classe `BigInteger`, para evitar a perda de precisão. Uma curiosidade: o fatorial de 50000 tem mais de 213.000 dígitos.

### 13.4.2 A classe `BigDecimal`

*classe* `BigDecimal`

A classe `BigDecimal` encapsula valores de ponto flutuante com precisão arbitrária, e contém vários métodos para efetuar operações com estes valores. As instâncias desta classe também são imutáveis - uma vez criadas, não podem ser modificadas, a não ser por uma nova associação entre uma instância e uma referência. Os métodos que implementam operações sempre retornam novas instâncias da classe. Nesta seção veremos os métodos mais úteis da classe e um exemplo mais completo de seu uso.

*escala*

Um conceito importante relacionado com a representação de valores na classe `BigDecimal` é a *escala* destes valores, ou seja, o número de casas decimais que a instância da classe usa para representar os valores - um valor sempre positivo. Em geral o programador não deverá se preocupar com a escala, que será automaticamente ajustada de acordo com a necessidade, mas quando uma operação que pode causar perda de precisão for executada, uma política de arredondamento deverá ser escolhida pelo programador.

Como exemplo de escalas, consideremos os valores 3.12 e 2.0009, ambos representados como instâncias da classe `BigDecimal`. O primeiro valor tem escala 2, e o segundo, 4. Caso criemos uma instância da classe com o resultado da soma ou da multiplicação dos dois valores, a escala do resultado será ajustada automaticamente para representar o resultado da operação sem perda de precisão. A operação de divisão, no entanto, não pode determinar automaticamente o valor da escala para o resultado, neste caso o programador terá que definir o que deverá ser feito com a escala e com o arredondamento.

*construtores para a classe* `BigDecimal`

Diferentemente da classe `BigInteger`, a classe `BigDecimal` não tem campos estáticos que representam instâncias da classe que encapsulam valores frequentemente usados - instâncias tem que ser criadas através de seus construtores ou fábricas de instâncias. Um dos construtores para a classe aceita como argumento um valor do tipo `double`, e cria a instância contendo aquele valor. Valores infinitos ou `NaN` não podem ser usados. É importante notar que nem sempre o resultado pode ser o esperado, já que nem sempre o valor de um `double` é o declarado. Por exemplo, se o construtor for chamado com o argumento 6.02, o valor encapsulado será aproximadamente igual a 6.019999999999999573674358543939888477325439453125. Outra forma do construtor, que aceita uma instância da classe `String` como argumento, pode ser mais útil. Esta forma do construtor causará a ocorrência de uma exceção `NumberFormatException` se a string não conter um valor válido na base 10.

Também podemos criar instâncias da classe `BigDecimal` usando instâncias da classe `BigInteger`. Um dos construtores aceita como argumento uma instância da classe `BigInteger` e outro aceita o mesmo argumento e mais um valor que será usado como escala inicial. Assim, podemos criar uma instância da classe `BigDecimal` contendo o valor 6.02 usando um construtor que aceita um `BigInteger` igual a 602 e a escala 2.

*método* `valueOf`

Novas instâncias também podem ser criadas com o método estático `valueOf`, que aceita como argumento um valor do tipo `long` e opcionalmente uma escala (tipo `int`), e usa estes va-

lores para a criação da nova instância. Diferentemente do construtor que aceita valores do tipo `double`, este método representa fielmente os valores passados: a instância criada como `new BigDecimal(602L,2)` representará exatamente 6.02.

*método* `scale` *método* `setScale`

O método `scale` retorna a escala da instância da classe `BigDecimal` como um valor do tipo `int`. É possível modificar a escala de uma instância da classe já criada, com o método `setScale`, que retorna uma nova instância da classe e que tem duas formas: uma que recebe um argumento do tipo inteiro representando a escala que será usada para criar a nova instância e outro que recebe adicionalmente um valor correspondente a um modo de arredondamento.

*modos de arredondamento*

Para compreender melhor os modos de arredondamento devemos entender que acontece algumas operações envolvendo instâncias de `BigDecimal` podem causar uma perda de precisão. A modificação da escala é uma destas operações: se temos uma instância da classe que representa o valor 1210.08 (cuja escala é 2), modificar a escala desta instância para qualquer valor menor do que dois causaria uma perda de precisão. Por outro lado, a escala da instância que encapsula o valor 717.0000 poderia ser modificada de quatro para zero sem perda de precisão. Se uma operação puder causar perda de precisão, um modo de arredondamento deve ser escolhido para ser usado. Existem oito modos de arredondamento, representados como campos estáticos finais da classe `BigDecimal` e sumarizados a seguir.

O modo de arredondamento `BigDecimal.ROUND_CEILING` arredonda os valores aproximando-os de infinito positivo, ou seja, “para cima”. Assim, se modificássemos a escala do valor 1.011 para 2 usando este modo, o valor resultante seria 1.02. O modo `BigDecimal.ROUND_FLOOR` arredonda aproximando de infinito negativo, ou seja, “para baixo”, no exemplo dado acima o valor final seria arredondado para 1.01.

O modo de arredondamento `BigDecimal.ROUND_DOWN` arredonda aproximando os valores de zero, simplesmente cortando o(s) dígito(s) a serem arredondandos. Este modo funciona de maneira idêntica ao modo `ROUND_FLOOR` se os valores forem positivos ou ao modo `ROUND_CEILING` se os valores forem negativos. O modo `BigDecimal.ROUND_UP` arredonda distanciando os valores do zero, funcionando da mesma forma que o modo `ROUND_CEILING` se os valores forem positivos ou igual ao modo `ROUND_FLOOR` se os valores forem negativos.

Outros três modos de arredondamento consideram a fração que será eliminada no arredondamento. O modo `BigDecimal.ROUND_HALF_DOWN` arredonda os valores para o vizinho mais próximo ou zero se os vizinhos forem equidistantes, de forma que se a escala do valor 1.015 for modificada para 2 usando este modo, o resultado será 1.01. O modo `BigDecimal.ROUND_HALF_UP` também arredonda para o vizinho mais próximo, mas se os vizinhos forem equidistantes, arredondará para o valor mais distante de zero. Assim, se a escala do valor 1.015 for modificada para 2 usando este modo, o resultado será 1.02. O modo `BigDecimal.ROUND_HALF_EVEN` arredonda tentando aproximar o valor para o vizinho mais próximo, e se equidistante, para o vizinho cujo último dígito seja par, desta forma ambos os valores 1.015 e 1.025 seriam arredondados para 1.02 se as escalas fossem modificadas para 2.

Um último modo de arredondamento é o modo `BigDecimal.ROUND_UNNECESSARY`, que informa aos métodos cujos resultados podem causar perda de precisão que o arredondamento não se fará necessário. Este modo somente poderá ser usado caso o resultado realmente não sofra perda de precisão, caso contrário a exceção `ArithmeticException` ocorrerá. Por exemplo, podemos usar este método para modificar a escala de 1.000 de três para zero, mas não poderemos usar o método para fazer o mesmo com o valor 1.001, pois com este certamente perda de precisão irá ocorrer.

*método* `compareTo`

Uma instância da classe `BigDecimal` pode ser comparada com outra usando o método `compareTo`, que recebe como argumento uma instância da própria classe e retorna $-1$ se o valor encapsulado for menor que o passado como argumento, 0 se forem iguais e $+1$ se o valor encapsulado for maior que o passado como argumento. As seis operações básicas de comparação (==, !=, >=, >, <= e <) podem ser simuladas comparando-se o resultado deste método com zero. Como exemplo, se duas instâncias da classe forem criadas como `BigDecimal x = new BigDecimal("3.1416"); BigDecimal y = new BigDecimal("3.14159");` o resultado da operação `x.compareTo(y)` será 1.

*método* `equals`

O método compara o valor encapsulado na instância com o valor passado como argumento (também uma instância de `BigDecimal`), retornando o valor booleano `true` somente se os dois valores forem exatamente iguais, até mesmo na escala. Por exemplo, considerando `b1` e `b2` como duas instâncias da classe `BigDecimal`, após o trecho de código `b1 = new BigDecimal("1.4142"); b2 = new BigDecimal("1.414200");`, o resultado de `b1.equals(b2)` será `false` pois as escalas de `b1` e `b2` serão diferentes, mesmo que os valores encapsulados sejam iguais (para este exemplo, o método `compareTo` retornaria 0).

*método* `max`

*método* `min`

Dois outros métodos de comparação de instâncias da classe `BigDecimal` são o método `max`, que recebe um argumento e retorna o maior dos valores (comparando o da instância que executa o método com o passado como argumento) e o método `min`, que retorna o menor dos valores. Ambos os método aguardam uma instância da classe `BigDecimal` como argumento.

*método* `add`

O método `add` adiciona o valor passado como argumento ao valor encapsulado na instância que executar o método, retornando uma nova instância da classe. Por exemplo, o trecho de código `BigDecimal saldo,novoSaldo; saldo = new BigDecimal("100000000"); novoSaldo = saldo.add(new BigDecimal("0.01"));` fará com que a instância `novoSaldo` contenha o valor 100000000.01. A instância resultante de uma chamada ao método `add` terá como escala a maior das escalas dos operadores envolvidos, assim se os valores 10.0000 e 91.45 forem somados, a escala do resultado será 4.

*método* `subtract`

O método `subtract` funciona da mesma maneira, subtraindo do valor encapsulado o que for passado como argumento. A escala do resultado será definida da mesma maneira que para o método `add`.

*método* `multiply`

O método `multiply` multiplicará o valor encapsulado pelo passado como argumento resultando em uma nova instância da classe. A escala será determinada como a soma das escalas dos valores sendo multiplicados, para garantir a precisão dos resultados. Assim, a execução do trecho de código `BigDecimal op1 = new BigDecimal("0.0001"); BigDecimal op2 = op1.multiply(new BigDecimal("0.09"));` fará com que `op2` represente 0.000009 (escala 6).

*método* `divide`

O método `divide` faz a divisão do valor encapsulado pelo passado como argumento, retornando uma nova instância da classe `BigDecimal`. Diferentemente dos outros métodos que fazem operações básicas, não é possível determinar, para a divisão, qual deve ser a escala do resultado, assim o programador deverá especificar explicitamente a escala e/ou o modo de arredondamento. Este método tem duas formas sobrecarregadas: uma que aceita um segundo argumento que é o modo de arredondamento, e outro que aceita a escala desejada e o modo de arredondamento. Exemplos de uso do método esclarecerão melhor este ponto: o trecho de código `BigDecimal valor, resultado; valor = new BigDecimal("10.00"); resultado = valor.divide(new BigDecimal("7"),BigDecimal.ROUND_CEILING);` divide 10 por 7, ar-

redondando para cima, resultando em 1.43 - a escala do dividendo foi usada. Em contraste, o trecho `BigDecimal valor,resultado; valor = new BigDecimal("10.00"); resultado = valor.divide(new BigDecimal("7"),15,BigDecimal.ROUND_CEILING);` divide 10 por 7, arredondando para cima e fazendo com que o resultado tenha 15 posições decimais (escala), resultando em 1.428571428571429.

O método `negate` (sem argumentos) funciona da mesma forma que o método `negate` da classe `BigInteger`, retornando uma nova instância da classe `BigDecimal` que é igual a $-1$ multiplicado pelo valor da instância que executar o método. A escala da instância retornada será a mesma da instância que executar o método. O método `abs` (também sem argumentos) retornará uma instância da classe sem o sinal, isto é, sempre positiva, e com a mesma escala. O método `signum` (também sem argumentos) retornará o sinal da instância como um valor do tipo `int` que será igual a $-1$, 0 ou $+1$ se o valor encapsulado for negativo, zero ou positivo, respectivamente.

*método* `negate`

*método* `abs`

Podemos converter valores armazenados em instâncias da classe `BigDecimal` para vários tipos nativos de Java. Por exemplo, para obter o valor inteiro correspondente ao encapsulado, podemos executar o método `intValue`, que retornará os 32 bits menos significativos do `BigDecimal`, desconsiderando as casas decimais e possivelmente causando perda de informação e resultando em valores inesperados se o `BigDecimal` for maior do que o maior valor representável por um `int`. Similarmente, o método `longValue` retornará o valor encapsulado representado como um `long`, sujeito às mesmas condições de perda de precisão. Para mais informações, veja as descrições dos métodos de mesmo nome na classe `BigInteger` e seus exemplos, na seção 13.4.1.

*método* `intValue`

*método* `longValue`

Os valores encapsulados em uma instância de `BigDecimal` também podem ser convertidos para valores nativos de ponto flutuante, com algumas regras de conversão. O método `floatValue` retorna o valor encapsulado como um valor do tipo nativo `float` se a conversão puder ser feita - caso o valor encapsulado pela instância de `BigDecimal` seja maior do que os que puderem ser representados pelo valor do tipo `float`, o método retornará `Float.POSITIVE_INFINITY` ou `Float.NEGATIVE_INFINITY`, dependendo do sinal do valor encapsulado. Para conversão do valor encapsulado para um valor do tipo nativo `double` podemos usar o método `doubleValue`, que está sujeito às mesmas regras de conversão: caso o valor encapsulado pela instância seja maior que o maior valor representável pelo tipo nativo, uma das constantes infinitas será retornada. A conversão de instâncias da classe `BigDecimal` para valores dos tipos `float` ou `double` pode causar outras perdas de precisão, pois `float`s e `double`s tem precisão limitada.

*método* `floatValue`

*método* `doubleValue`

Outro método considerado de conversão é o método `toString`, que funciona como os métodos `toString` de diversas classes que já foram estudadas. Este método retorna uma representação da instância de `BigDecimal` como uma string.

*método* `toString`

Um último método de conversão de interesse é o método `toBigInteger`, que converte a instância de `BigDecimal` para uma instância de `BigInteger`, descartando as casas decimais mas sem nenhuma outra perda de informação ou precisão.

*método* `toBigInteger`

Um exemplo de uso da classe `BigInteger` é mostrado a seguir. Consideremos o seguinte algoritmo iterativo de cálculo da raiz quadrada de um valor qualquer, que usa aproximações, um erro máximo admissível e recursividade. Os passos do algoritmo são os seguintes:

1. Para calcular a raiz quadrada do valor $N$ com um erro máximo $E$, usamos um valor que será a aproximação inicial da raiz, chamado $A$.
2. Calculamos o valor absoluto da diferença entre $A^2$ e $N$. Se este valor for menor que o erro $E$, retornamos $A$ como sendo a raiz aproximada.

3. Se o valor absoluto da diferença for maior que $E$, recalculamos $A$ como $(A^2+N)/(2A)$ e voltamos ao passo 2.

A classe `BigMath` contém um método estático que calcula a raiz quadrada de um valor usando o algoritmo acima e instâncias da classe `BigDecimal` para armazenar os valores e fazer os cálculos. A classe `BigMath` é mostrada na listagem 13.9.

Listagem 13.9: A classe `BigMath`, que contém um método estático que calcula a raiz quadrada de um número usando aproximações, recursão e instâncias da classe `BigDecimal`.

```
1  /* Precisaremos da classe BigDecimal que está no pacote java.math, importaremos
2     todas as classes deste pacote. */
3  import java.math.*;
4
5  /**
6   * A classe BigMath, que contém um método estático que calcula a raiz quadrada
7   * de um número usando aproximações e recursão. A raiz quadrada é calculada usando
8   * instâncias da classe BigDecimal para armazenar os resultados, assim a precisão
9   * dos resultados será quase arbitrária.
10  */
11 class BigMath // declaração da classe
12   {
13  /**
14   * O método raizQuadrada calcula a raiz quadrada de um valor qualquer, usando uma
15   * aproximação inicial, um erro máximo admissível e um algoritmo recursivo. A raiz
16   * será calculada com pelo menos a precisão especificada pelo erro, isto é, a
17   * diferença entre o valor real e o calculado será menor que o erro.
18   * @param valor o valor para o qual calcularemos a raiz quadrada
19   * @param aproximação a aproximação obtida até agora para a raiz quadrada
20   * @param erro o erro máximo admissível para este cálculo
21   * @return a raiz quadrada do valor, obtida através de aproximações
22   */
23   public static BigDecimal raizQuadrada(BigDecimal valor,BigDecimal aproximação,
24                                         BigDecimal erro)
25     {
26     // Calculamos a diferença entre o quadrado da aproximação e o valor.
27     BigDecimal diferença = aproximação.multiply(aproximação).subtract(valor);
28     // Se o valor absoluto da diferença entre a aproximação^2 e o valor for menor
29     // que o erro, consideramos a aproximação boa o suficiente e a retornamos.
30     if (diferença.abs().compareTo(erro) < 0) return aproximação;
31     // Se não, calculamos a nova aproximação como sendo (aproximação^2+valor) dividido
32     // por (2*aproximação) e executamos novamente o algoritmo. Primeiro calculamos
33     // os termos da divisão.
34     BigDecimal dividendo = aproximação.multiply(aproximação).add(valor);
35     BigDecimal divisor = aproximação.multiply(new BigDecimal("2"));
36     // Como a divisão possivelmente fará com que o resultado tenha que ter uma escala
37     // maior que a do divisor ou dividendo, usamos a maior escala destes mais um como
38     // a nova escala.
39     int novaEscala = Math.max(dividendo.scale(),divisor.scale())+1;
40     // Calculamos a nova aproximação
41     BigDecimal novaAproximação =
42       dividendo.divide(divisor,novaEscala,BigDecimal.ROUND_HALF_DOWN);
43     // Executamos a chamada recursiva com a nova aproximação.
44     return raizQuadrada(valor,novaAproximação,erro);
45     }
46
47   } // fim da classe BigMath
```

O método `raizQuadrada` da classe `BigMath` é demonstrado pela classe `DemoBigMath`, mostrada na listagem 13.10.

Listagem 13.10: A classe `DemoBigMath`, que demonstra o uso do método `raizQuadrada` da classe `BigMath`.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
/* Precisaremos da classe BigDecimal que está no pacote java.math, importaremos
   todas as classes deste pacote. */
import java.math.*;

/**
 * A classe DemoBigMath, que contém um método estático que calcula a raiz quadrada
 * de um número usando aproximações e recursão. A raiz quadrada é calculada usando
 * instâncias da classe BigDecimal para armazenar os resultados, assim a precisão
 * dos resultados será quase arbitrária.
 */
class DemoBigMath // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método chama o método
  * raizQuadrada com vários valores como argumentos, demonstrando a precisão que
  * pode ser obtida.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    BigDecimal dois = new BigDecimal("2");
    BigDecimal um = new BigDecimal("1");
    BigDecimal primeiraPrecisão = new BigDecimal("0.1");
    BigDecimal segundaPrecisão  = new BigDecimal("0.00000001");
    BigDecimal terceiraPrecisão = new BigDecimal("0.00000000000000000000000000000001");
    System.out.println(BigMath.raizQuadrada(dois,um,primeiraPrecisão));
    System.out.println(BigMath.raizQuadrada(dois,um,segundaPrecisão));
    System.out.println(BigMath.raizQuadrada(dois,um,terceiraPrecisão));
    // Para comparação, vamos ver o que o pacote Math diz
    System.out.println(Math.sqrt(2));
    }

  } // fim da classe DemoBigMath
```

## 13.5 Exercícios do capítulo 13

**Exercício 13.1:** ⋆

Qual das seguintes chamadas a construtores para a classe `BigInteger` serão válidas ?

**A.** `new BigInteger("0123456789");`
**B.** `new BigInteger(" 1 ");`
**C.** `new BigInteger("-10+15");`
**D.** `new BigInteger("ABCDEF",16);`
**E.** `new BigInteger("ABCDEF","16");`

**Exercício 13.2:** ⋆⋆

Modifique a implementação do algoritmo de localização de uma raiz de uma função, implementado na classe `RaizDeFuncao` (listagem 13.4), para que este detecte quando os sinais dos extremos do intervalo forem iguais (o que impossibilita a aplicação do algoritmo).



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 13.3:** ⋆⋆

O comprimento do arco $ABC$ de uma parábola é dada por $\frac{1}{2}\sqrt{b^2+16a^2} + \frac{b^2}{8a} + log\left(\frac{4a+\sqrt{b^2+16a^2}}{b}\right)$, onde $a$ é a altura da parábola e $b$ é a distância entre $A$ e $C$ (veja a figura 13.5). Escreva um programa em Java que leia os valores $a$ e $b$ e calcule o comprimento do arco da parábola correspondente.

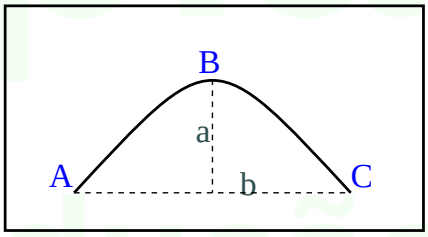



Figura 13.5: Medidas de uma parábola.

**Exercício 13.4:** ⋆⋆

Escreva um programa que resolva uma equação do segundo grau. Este programa deve ler os valores $A$, $B$ e $C$ do teclado, calcular o valor $\Delta = B^2 - 4AC$ e as raízes $x_1 = \frac{-B-\sqrt{\Delta}}{2A}$ e $x_2 = \frac{-B-\sqrt{\Delta}}{2A}$. *Dica:* se $\Delta$ for menor que zero, não existem raízes reais para a equação - o programa deve prever este caso e imprimir uma mensagem correspondente.

**Exercício 13.5:** ⋆⋆

Implemente a operação $x^2 - 3x + 9$ usando instâncias da classe `BigInteger`.

**Exercício 13.6:** ⋆⋆

Implemente a operação $3x^4 - 2x^3 + x^2 - 2x + 19$ usando instâncias da classe `BigInteger`.

**Exercício 13.7:** ⋆⋆

Implemente a operação $(x + 2y - 3z) * (2x - 4y + z)$ usando instâncias da classe `BigInteger`.

**Exercício 13.8:** ⋆⋆

Implemente a operação $(1+x)/(1-x)$ usando instâncias da classe `BigDecimal`.

**Exercício 13.9:** ⋆⋆

Implemente a operação $(1-2x)/(x^3-4x)$ usando instâncias da classe `BigDecimal`.

**Exercício 13.10:** ⋆⋆

Implemente a operação $(2x+y+3z)/(2x*2y*2z)$ usando instâncias da classe `BigDecimal`.

**Exercício 13.11:** ⋆⋆

Escreva a classe `Dado`, que encapsula o funcionamento de um dado, com um método estático que retorne um valor pseudo-aleatório entre 1 e 6 cada vez que for executado.

**Exercício 13.12:** ⋆⋆

Escreva a classe `DadoViciado`, que encapsula o funcionamento de um dado viciado, que faz com que o número um saia (teoricamente) duas vezes mais do que os outros números. Esta classe deve ter um método estático que retorne o valor sorteado pelo dado cada vez que for executado.

**Exercício 13.13:** ⋆⋆⋆

O algoritmo usado para implementar o método de localização de uma raiz de uma função, implementado na classe `RaizDeFuncao` (listagem 13.4) considera como pontos de parada a localização da raiz exata (o que nem sempre pode ocorrer) ou o estreitamento do intervalo de busca. Uma terceira condição possível de parada é a localização de um valor para o qual a função retorne um valor próximo de zero mas não necessariamente zero: por exemplo, se uma função qualquer retornar o valor 0.00000001 para um argumento passado para ela, podemos considerar que este argumento é uma raiz com aproximação boa o suficiente. Modifique a implementação do algoritmo para que um valor adicional $\varepsilon$ seja especificado, e se o valor calculado para a função for menor ou igual a este valor $\varepsilon$ o algoritmo considere que uma raiz foi achada.

**Exercício 13.14:** ⋆⋆⋆

Reescreva o método `main` da classe `RaizDeFuncao` (listagem 13.4) para que seja um método estático que possa ser chamado de qualquer classe. Faça com que este método calcule a raiz da função de forma recursiva.

**Exercício 13.15:** ⋆⋆⋆

Escreva um método na classe `BigMath` (listagem 13.9) que calcule o máximo divisor comum entre duas instâncias da classe `BigInteger` usando o algoritmo de Euclides (veja o exercício 7.26). A classe`BigInteger` já tem um método para isto, tente implementar uma versão recursiva do algoritmo apresentado.

**Exercício 13.16:** ⋆⋆⋆

Escreva uma versão recursiva para o método `fatorial` (que utiliza instâncias da classe `BigInteger`). Veja as classes `BigFatorial` (listagem 13.8) e `BigMath` (listagem 13.9) como referência.

**Exercício 13.17:** ⋆⋆⋆

Reescreva a classe `ProbabilidadeBasica` (listagem 7.7) para que todos os seus métodos usem instâncias da classe `BigInteger` para cálculos. Veja também a listagem 13.8.

**Exercício 13.18:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $1/2$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.37 para informações sobre o cálculo da sequência.

**Exercício 13.19:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um um método que calcule a sequência que converge para $3/4$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.38 para informações sobre o cálculo da sequência.

**Exercício 13.20:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para o logaritmo de 2 usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.40 para informações sobre o cálculo da sequência.

**Exercício 13.21:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $e^x$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.55 para informações sobre o cálculo da sequência. *Dica:* use o método `fatorial` mostrado na classe `BigFatorial` (listagem 13.8).

**Exercício 13.22:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $\pi$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.53 para informações sobre o cálculo da sequência.

**Exercício 13.23:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $\pi$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.54 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.24:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule $sin(x)$ usando séries e instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.56 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante $\pi$ calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.25:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $sin(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.57 para informações sobre outra maneira de calcular a sequência. *Dica:* use o método `fatorial` mostrado na classe `BigFatorial` (listagem 13.8).

**Exercício 13.26:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $cos(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.60 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante $\pi$ calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.27:** ★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $cos(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.61 para informações sobre outra maneira de calcular a sequência. *Dica:* use o método `fatorial` mostrado na classe `BigFatorial` (listagem 13.8).

**Exercício 13.28:** ★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $tan(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.65 para informações sobre o cálculo da sequência. *Dica:* use o método `fatorial` mostrado na classe `BigFatorial` (listagem 13.8).

**Exercício 13.29:** ★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $cotag(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.67 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante $\pi$ calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.30:** ★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $sec(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.69 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante $\pi$ calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.31:** ★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $cosec(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.71 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante $\pi$ calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.32:** ★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $sinh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.58 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante $\pi$ calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.33:** ★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $sinh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.59 para informações sobre outra maneira de calcular a sequência. *Dica:* use o método `fatorial` mostrado na classe `BigFatorial` (listagem 13.8).

**Exercício 13.34:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $cosh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.62 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante π calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.35:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $cosh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.63 para informações sobre outra maneira de calcular a sequência. *Dica:* use o método `fatorial` mostrado na classe `BigFatorial` (listagem 13.8).

**Exercício 13.36:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $tanh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.66 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante π calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.37:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $cotanh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.68 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante π calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.38:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $sech(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.70 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante π calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.39:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $cosech(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.72 para informações sobre o cálculo da sequência. *Dica:* o método deverá usar o valor da constante π calculada como um valor encapsulado em um `BigDecimal`, use o resultado do exercício 13.22 ou 13.23.

**Exercício 13.40:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $arctanh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.73 para informações sobre o cálculo da sequência.

**Exercício 13.41:** ⋆⋆⋆

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $arccotanh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.74 para informações sobre o cálculo da sequência.

**Exercício 13.42:** ⋆⋆⋆

Escreva uma aplicação em Java que sorteie 50 números entre 0 e 99 para jogar na LotoMania. A aplicação não deverá sortear duas vezes o mesmo número. *Dica:* usar um array para armazenar os valores já sorteados pode facilitar a solução do problema.

**Exercício 13.43:** ⋆⋆⋆

Outra maneira de resolver o exercício 13.42 é criar um array onde cada elemento vale o valor do seu índice (assim `array[0]` será igual a 0, `array[1]` será igual a 1 e assim sucessivamente). Este array pode ser *desordenado* se escolhermos dois índices entre 0 e 99 e trocarmos os elementos deste índice, e repetirmos este procedimento um número suficiente de vezes. Ao final da desordenação do array, basta pegar os primeiros cinquenta números como resultado do algoritmo.

**Exercício 13.44:** ⋆⋆⋆

Modifique a resposta do exercício 13.11 para que um histórico seja criado e atualizado para os lançamentos do dado - cada vez que um valor for sorteado, o histórico deverá incrementar um contador adequado. Escreva, para a classe modificada, um método `imprimeHistórico` que imprima o histórico de arremessos. Escreva uma aplicação que use a classe modificada, lançando o dado alguns milhões de vezes, para ver se os números saem aproximadamente com a mesma frequência.

**Exercício 13.45:** ⋆⋆⋆

Repita o exercício 13.44 mas usando o dado viciado (exercício 13.12). Escreva uma aplicação que use a classe modificada, lançando o dado alguns milhões de vezes, para ver a distribuição da saída dos valores.

**Exercício 13.46:** ⋆⋆⋆

Podemos otimizar algumas classes e aplicações que requerem o cálculo do seno e coseno de muitos ângulos através do cálculo preliminar dos senos e cosenos dos ângulos que serão usados e armazenando estes resultados em arrays. Desta forma, ao invés de calcular as funções cada vez que se fizerem necessárias, basta acessar arrays (um para cada função) que já terão os valores armazenados. Esta técnica funcionará melhor se os ângulos que forem usados forem valores inteiros (expressos em graus). Escreva uma classe `TabelaDeFuncoes` cujo construtor declare dois arrays do tipo `double` de 360 posições cada e preencha estes arrays com os valores dos senos e cosenos dos ângulos em graus indicados pelos índices. Escreva nesta classe os métodos para recuperar os valores dos senos e cosenos - estes métodos devem receber o valor inteiro do ângulo em graus.

**Exercício 13.47:** ★★★★

O algoritmo usado para cálculo das raízes da função implementado na classe `RaizDeFuncao` (listagem 13.4) nem sempre funciona como esperado: mesmo que as condições iniciais sejam satisfeitas (isto é, os sinais dos dois extremos dos intervalos de busca sejam diferentes) o algoritmo pode encontrar uma raiz quando mais de uma existirem no mesmo intervalo. Um exemplo pode ser dado se procurarmos a raiz da função $x^3 - 3x^2 + x$ entre $-1$ e 5: existem três raízes no intervalo, mas o algoritmo somente localizará uma. Modifique o algoritmo implementando uma *heurística* que possibilita a procura de mais de uma raiz, fazendo com que os intervalos descartados sejam procurados novamente. Por exemplo, para a função mostrada, o ponto central será 2 e o intervalo $-$1-2 será descartado, pois os sinais das suas funções serão iguais. Faça com que o algoritmo procure um valor neste intervalo descartado tentando localizar um valor para o qual o sinal seja diferente dos sinais dos extremos. *Dica:* tente usar uma solução recursiva para este problema.

**Exercício 13.48:** ★★★★

O cálculo da somatória na classe `RegraDoTrapezio` pode ser considerado ineficiente: cada laço calculará $f(x_n)$ e $f(x_{n+1})$, e na próxima iteração, $f(x_{n+1})$ e $f(x_{n+2})$, sendo que $f(x_{n+1})$ já tinha sido calculado. Otimize o cálculo de forma que subcálculos redundantes não sejam feitos. Meça a performance das duas versões para certificar-se da otimização. Verifique também se existem outros pontos que podem ser otimizados.

**Exercício 13.49:** ★★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência que converge para $ln(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.85 para informações sobre o cálculo da sequência.

**Exercício 13.50:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $ln(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.90 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.51:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arcsen(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.84 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.52:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arccos(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.87 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.53:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arcsec(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.89 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.54:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arccosec(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.88 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.55:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arctan(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.94 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.56:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arccotan(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.95 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.57:** ★★★★

Escreva, para a classe `BigMath`, um outro método que calcule a sequência que converge para $arcsenh(x)$ usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.96 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.58:** ★★★★

Escreva, para a classe `BigMath`, um método que calcule a sequência de Euler, que converge para $\pi$, usando instâncias de `BigDecimal` para o cálculo. Veja o exercício 7.92 para informações sobre outra maneira de calcular a sequência.

**Exercício 13.59:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor de $\pi$ com as fórmulas mostradas no enunciado do exercício 7.108. Compare a convergência das diversas fórmulas.

**Exercício 13.60:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor de $\pi$ com as fórmulas de Machin mostradas no enunciado do exercício 7.109. Compare a convergência das diversas fórmulas.

**Exercício 13.61:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor de $1/\pi$ com as fórmulas mostradas no enunciado do exercício 7.105. Compare a convergência das diversas fórmulas.

**Exercício 13.62:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem indiretamente o valor de $\pi$ com as fórmulas mostradas no enunciado do exercício 7.106. Compare a convergência das diversas fórmulas.

**Exercício 13.63:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem indiretamente o valor da raiz quadrada de 2 com as fórmulas mostradas no enunciado do exercício 7.110. Compare a convergência das diversas fórmulas.

**Exercício 13.64:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor do logaritmo natural de 2 com as fórmulas mostradas no enunciado do exercício 7.107. Compare a convergência das diversas fórmulas.

**Exercício 13.65:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor do logaritmo natural de 2 com as fórmulas de Machin mostradas no enunciado do exercício 7.111. Compare a convergência das diversas fórmulas.

**Exercício 13.66:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor de $\gamma$ (a cosntante de Euler) com as fórmulas mostradas no enunciado do exercício 7.112. Compare a convergência das diversas fórmulas.

**Exercício 13.67:** ★★★★★

Escreva, para a classe `BigMath`, métodos que calculem o valor de $e$ (base dos logaritmos naturais) com as fórmulas mostradas no enunciado do exercício 7.113. Compare a convergência das diversas fórmulas.

## 13.6 Exercícios do capítulo 13 que envolvem cálculo de funções

A criação de classes executáveis que envolvem o cálculo de funções reforçam o aprendizado dos métodos de Java apresentados neste capítulo e da forma com que podemos expressar funções matemáticas com métodos da classe `Math`. Nesta seção vários exercícios que pedem a expressão de funções matemáticas em Java (como parte do cálculo de integrais ou de localização de raízes das funções) são apresentados.

Alguns dos exercícios que envolvem integrais pedem que a integral seja calculada entre zero e um valor qualquer. O valor zero exato não deve ser usado para evitar que durante o cálculo ocorra uma divisão por zero, o que resultaria em infinito. Nestes casos, deve-se usar um valor próximo de zero.

Alguns dos exercícios sobre integrais neste capítulo foram retirados do livro *Manual de Fórmulas e Tabelas Matemáticas*, de Murray R. Spiegel, da coleção Schaum da editora McGraw-Hill (1973). Muitos destes foram verificados e adequados por Nadia Maria França Borges.

**Exercício 13.68:** ★

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^2(\frac{1}{x})\partial x$ usando 100 subintervalos.

**Exercício 13.69:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_2^{10}(\frac{1}{1+m})\partial m$ usando 200 subintervalos.

**Exercício 13.70:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-5}^{5}(y^2+3y+1)\partial y$ usando 1000 subintervalos.

**Exercício 13.71:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^4(2+g-g^2)\partial g$ usando 400 subintervalos.

**Exercício 13.72:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^4(x+\frac{1}{x})\partial x$ usando 300 subintervalos.

**Exercício 13.73:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_4^8(\frac{10}{a^2+1})\partial a$ usando 1600 subintervalos.

**Exercício 13.74:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^2(\frac{2w-3}{3w})\partial w$ usando 200 subintervalos.

**Exercício 13.75:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-1}^{1}(z^3-3z+2)\partial z$ usando 100000 subintervalos.

**Exercício 13.76:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-3}^{3}(2x^4-3x^3-5x^2+6x-10)\partial x$ usando 700 subintervalos.

**Exercício 13.77:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^2(\frac{1}{t(t^2+4)})\partial t$ usando 500 subintervalos.

**Exercício 13.78:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^2(s\sqrt{s^2-1})\partial s$ usando 2000 subintervalos.

**Exercício 13.79:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^2(\frac{1}{p\sqrt{p^4+1}})\partial p$ usando 40000 subintervalos.

**Exercício 13.80:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{1}{\sqrt{1+w^2}})\partial w$ usando 1000 subintervalos.

**Exercício 13.81:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\sqrt{1+q^2})\partial q$ usando 100000 subintervalos.

**Exercício 13.82:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^4(\frac{1+y}{y^2})\partial y$ usando 9000 subintervalos.

**Exercício 13.83:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-3}^0(\frac{1}{\sqrt{25+3h}})\partial h$ usando 60000 subintervalos.

**Exercício 13.84:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\sqrt{2}/2}(\frac{1}{\sqrt{1+x^2}})\partial x$ usando 80000 subintervalos.

**Exercício 13.85:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{y^2}{\sqrt{y^6+4}})\partial y$ usando 10000 subintervalos.

**Exercício 13.86:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_2^{3.5}(\frac{1}{\sqrt{5+4p+p^2}})\partial p$ usando 150000 subintervalos.

**Exercício 13.87:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^2(w^{2/3})\partial w$ usando 200 subintervalos.

**Exercício 13.88:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(e^{r^2})\partial r$ usando 100 subintervalos.

**Exercício 13.89:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1 (p^{e/2}) \partial p$ usando 50000 subintervalos.

**Exercício 13.90:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1 (\frac{1}{1+3e^{2x}}) \partial x$ usando 250000 subintervalos.

**Exercício 13.91:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-1}^1 (2^z) \partial z$ usando 40000 subintervalos.

**Exercício 13.92:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-1}^1 (1.08^{g^3}) \partial g$ usando 200 subintervalos.

**Exercício 13.93:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-5}^5 (k^8 - 6k^7 + 2k^5 - k^4 + 9k^3 + 2k^2 - k - 1) \partial k$ usando 25000 subintervalos.

**Exercício 13.94:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_1^{10} (ln(y)) \partial y$ usando 900000 subintervalos.

**Exercício 13.95:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_2^3 (ln(1+x^2)) \partial x$ usando 1000 subintervalos.

**Exercício 13.96:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_2^{e^2} (\frac{1}{m\ ln(m)}) \partial m$ usando aproximadamente $100e^2$ subintervalos.

**Exercício 13.97:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi} (sen(z)) \partial z$ usando aproximadamente $10000\pi$ subintervalos.

**Exercício 13.98:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi} (cos(z)) \partial z$ usando aproximadamente $100\pi$ subintervalos.

**Exercício 13.99:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{\pi}^{2\pi}(sen(s)/2)\partial s$ usando aproximadamente $100000\pi$ subintervalos.

**Exercício 13.100:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{\pi}^{2\pi/3}(\frac{sen(t/2)}{sen(t/2)cos(t/2)})\partial t$ usando 1000 subintervalos.

**Exercício 13.101:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-\pi/2}^{\pi/2}((x^2-3x)sen(5x))\partial x$ usando aproximadamente $500\pi$ subintervalos.

**Exercício 13.102:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{0}^{\pi/2}(\frac{\pi}{4}sen(a)-\frac{\pi}{4}cos(a))\partial a$ usando aproximadamente $100000\pi$ subintervalos.

**Exercício 13.103:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{0}^{\pi/2}(\frac{1}{2sen(w)+3cos(w)+5})\partial w$ usando aproximadamente $1000\pi$ subintervalos.

**Exercício 13.104:** ⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_{-1}^{1}(e^x sen(x)sen(3x))\partial x$ usando 1000 subintervalos.

**Exercício 13.105:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $x^3-2x+8$ procurando raízes no intervalo entre $-4$ e $-2$.

**Exercício 13.106:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $-x^5+16x^4-4x^3+32x^2+256x+1024$ procurando raízes no intervalo entre 10 e 20.

**Exercício 13.107:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $e^{x/25}-e^{x/3}+1$ procurando raízes no intervalo entre $-10$ e 10.

**Exercício 13.108:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $e^{4sin(x)}-5$ procurando raízes no intervalo entre $-5$ e 0.

**Exercício 13.109:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $log(2/x)$ procurando raízes no intervalo entre 1 e 10.

**Exercício 13.110:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $log(-x^3+8x^2+16x+256)-5$ procurando raízes no intervalo entre 10 e 11.

**Exercício 13.111:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $log(2sin(x))$ procurando raízes no intervalo entre 1 e 3.

**Exercício 13.112:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $sin(x)*cos(2/x)$ procurando raízes no intervalo entre $-2$ e $-1$.

**Exercício 13.113:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $sin(x^2+x+2)$ procurando raízes no intervalo entre $-1$ e 1.5.

**Exercício 13.114:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $cos(2x)+2cos(x)$ procurando raízes no intervalo entre 0 e 2.

**Exercício 13.115:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $tan(x+3)$ procurando raízes no intervalo entre $-1$ e 1.

**Exercício 13.116:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $sin(2/x^4)-0.5$ procurando raízes no intervalo entre 0 e 1.1.

**Exercício 13.117:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $sin(x)*sin(x^2)-0.1$ procurando raízes no intervalo entre $-3.5$ e $-2$.

**Exercício 13.118:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $0.35+cos(x)/(2+cos(2x)$ procurando raízes no intervalo entre 2 e 3.

**Exercício 13.119:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $0.5-atan(x+0.5)$ procurando raízes no intervalo entre $-1$ e 1.

**Exercício 13.120:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $acos(x) - 2$ procurando raízes no intervalo entre $-0.5$ e 0.5.

**Exercício 13.121:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $atan(x^2 - 3x + x/2)$ procurando raízes no intervalo entre 1 e 5.

**Exercício 13.122:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $asin(x + 0.4)$ procurando raízes no intervalo entre $-0.5$ e 0.5.

**Exercício 13.123:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $\sqrt{1 + sin(x+1)} - 1$ procurando raízes no intervalo entre $-2$ e 2.

**Exercício 13.124:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $\sqrt{x^3} - \sqrt{31x^2}$ procurando raízes no intervalo entre 1 e 40.

**Exercício 13.125:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $abs(x^3 - 4) - 1$ procurando raízes no intervalo entre 0 e 1.6.

**Exercício 13.126:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $abs(sin(x)) - 0.5$ procurando raízes no intervalo entre $-2.5$ e 0.

**Exercício 13.127:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $x^{8/3} - 3x - 8$ procurando raízes no intervalo entre 0 e 10.

**Exercício 13.128:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $8^x - x^8$ procurando raízes no intervalo entre 0.5 e 2.

**Exercício 13.129:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $x^{3.2} - 2x^3 + x^{2.5}$ procurando raízes no intervalo entre 5 e 20.

**Exercício 13.130:** ⋆

Usando como base a classe `RaizDeFuncao` (listagem 13.4), escreva um programa que calcule a raiz da função $x^{-1.8} - x^{0.9} + x^{1.1} - 1$ procurando raízes no intervalo entre 2 e 4.

**Exercício 13.131:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^a (\frac{1}{\sqrt{a^2-z^2}})\partial z$, cujo resultado deve ser igual a $\frac{\pi}{2}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.132:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^a (\sqrt{a^2-w^2})\partial w$, cujo resultado deve ser igual a $\frac{\pi a^2}{4}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.133:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{\pi/2} (sin(x))^2 \partial x$ e $\int_0^{\pi/2} (cos(x))^2 \partial x$, cujos resultados devem ser iguais a $\frac{\pi}{4}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.134:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{2\pi} (\frac{1}{a+b\ sin(r)})\partial r$ e $\int_0^{2\pi} (\frac{1}{a+b\ cos(r)})\partial r$, cujos resultados devem ser iguais a $\frac{2\pi}{\sqrt{a^2-b^2}}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.135:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi/2} (\frac{1}{a+b\ cos(s)})\partial s$, cujos resultado deve ser igual a $\frac{arccos(b/a)}{\sqrt{a^2-b^2}}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.136:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{2\pi} (\frac{1}{(a+b\ sin(t))^2})\partial t$ e $\int_0^{2\pi} (\frac{1}{(a+b\ cos(t))^2})\partial t$, cujos resultados devem ser iguais a $\frac{2\pi\ a}{(a^2-b^2)^{3/2}}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.137:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{2\pi} (\frac{1}{1-2a\ cos(k)+a^2})\partial k$, cujos resultado deve ser igual a $\frac{2\pi}{1-a^2}$, para $0 < a < 1$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.138:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1 (\frac{arcsen(x)}{x})\partial x$, cujos resultado deve ser igual a $\frac{\pi}{2} ln(2)$. Varie o número de subintervalos para verificar a precisão do resultado.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 13.139:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{ln(x)}{1+x})\partial x$, cujos resultado deve ser igual a $-\frac{\pi^2}{12}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.140:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{ln(x)}{1-x})\partial x$, cujos resultado deve ser igual a $-\frac{\pi^2}{6}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.141:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{ln(1+y)}{y})\partial y$, cujos resultado deve ser igual a $\frac{\pi^2}{12}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.142:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{ln(1-y)}{y})\partial y$, cujos resultado deve ser igual a $-\frac{\pi^2}{6}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.143:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(ln(x)ln(1+x))\partial y$, cujos resultado deve ser igual a $2-2ln(2)-\frac{\pi^2}{12}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.144:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(ln(x)ln(1-x))\partial y$, cujos resultado deve ser igual a $2-\frac{\pi^2}{6}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.145:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{b^m-b^n}{ln(b)})\partial b$, cujos resultado deve ser igual a $ln\left(\frac{m+1}{n-1}\right)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.146:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi}(w\ ln(sen(w)))\partial w$, cujos resultados devem ser iguais a $-\frac{\pi^2}{2}ln(2)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.147:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi/2}(sen(z)ln(sen(z)))\partial z$, cujos resultado deve ser igual a $ln(2)-1$. Varie o número de subintervalos para verificar a precisão do resultado.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 13.148:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{\pi/2}(ln(sin(x)))\partial x$ e $\int_0^{\pi/2}(ln(cos(x)))\partial x$, cujos resultados devem ser iguais a $-\frac{\pi^2}{2}ln(2)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.149:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{\pi/2}(ln(sin(x))^2)\partial x$ e $\int_0^{\pi/2}(ln(cos(x))^2)\partial x$, cujos resultados devem ser iguais a $\frac{\pi^2}{2}ln(2)^2+\frac{\pi^3}{24}$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.150:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{2\pi}(ln(a+b\ sen(z)))\partial z$ e $\int_0^{2\pi}(ln(a+b\ cos(z)))\partial z$, cujos resultados devem ser iguais a $2\pi\ ln(a+\sqrt{a^2-b^2})$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.151:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi}(ln(a+b\ cos(j))\partial j$, cujo resultado deve ser igual a $\pi\ ln\left(\frac{a+\sqrt{a^2-b^2}}{2}\right)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.152:** ⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi/4}(ln(1+tan(q)))\partial q$, cujo resultado deve ser igual a $\frac{\pi}{8}ln(2)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.153:** ⋆⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{\pi/2}(\frac{1}{sin(j)})\partial j$, cujo resultado deve ser igual ao da série $2\left(\frac{1}{1^2}-\frac{1}{3^2}+\frac{1}{5^2}-\frac{1}{7^2}+\frac{1}{9^2}-\frac{1}{11^2}+\cdots\right)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.154:** ⋆⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^{1}(\frac{arctan(w)}{w})\partial w$, cujo resultado deve ser igual ao da série $\left(\frac{1}{1^2}-\frac{1}{3^2}+\frac{1}{5^2}-\frac{1}{7^2}+\frac{1}{9^2}-\frac{1}{11^2}+\cdots\right)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.155:** ⋆⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{\pi/2}(sin(x))^{2m}\partial x$ e $\int_0^{\pi/2}(cos(x))^{2m}\partial x$, cujos resultados devem ser iguais ao da série $\frac{1\times3\times5\times7\times\cdots2m-1}{2\times4\times6\times8\times\cdots2m}\frac{\pi}{2}$ para $m$ inteiro e maior que zero. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.156:** ⋆⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor das integrais definidas $\int_0^{\pi/2}(sin(x))^{2m+1}\partial x$ e $\int_0^{\pi/2}(cos(x))^{2m+1}\partial x$, cujos resultados devem ser iguais ao da série $\frac{2\times4\times6\times8\times\cdots2m}{1\times3\times5\times7\times\cdots2m-1}$ para $m$ inteiro e maior que zero. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.157:** ⋆⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^a(ln\left(2sen\left(\frac{w}{2}\right)\right))\partial w$, cujo resultado deve ser igual ao da série $-\left(\frac{sen(a)}{1^2}+\frac{sen(2a)}{2^2}+\frac{sen(3a)}{3^2}+\frac{sen(4a)}{4^2}+\cdots\right)$. Varie o número de subintervalos para verificar a precisão do resultado.

**Exercício 13.158:** ⋆⋆⋆

Usando como base a classe `RegraDoTrapezio` (listagem 13.3), escreva um programa que calcule o valor da integral definida $\int_0^1(\frac{1}{c^c})\partial c$, cujo resultado deve ser igual ao da série $\left(\frac{1}{1^1}+\frac{1}{2^2}+\frac{1}{3^3}+\frac{1}{4^4}+\cdots\right)$. Varie o número de subintervalos para verificar a precisão do resultado.

## 13.7 Exercícios complementares do capítulo 13

**Exercício 13.159:** ⋆

Modifique a classe `RegraDoTrapezio` (listagem 13.3) para que os valores dos intervalos inferior e superior e número de intervalos sejam entrados pelo teclado. Faça com que o programa verifique se o intervalo superior é maior que o intervalo inferior e mostre uma mensagem de erro se não for.

**Exercício 13.160:** ⋆⋆

Escreva a classe `PoligonoRegular`, que implementa a interface `ObjetoGeometrico` (listagem 9.5) e que represente um polígono regular de $N$ lados, com um ponto central. Use como referência as classes `Circulo` e `Retangulo` (listagens 9.6 e 9.7). *Dica:* o perímetro de um polígono regular de $n$ lados é dado por $n \times b$ onde $b$ é o comprimento do lado do polígono. Sua área é dada por $\frac{1}{4}nb^2\frac{cos(\pi/n)}{sin(\pi/n)}$. Veja também os exercícios 9.30 e 9.19.

**Exercício 13.161:** ⋆⋆

Escreva a classe `Elipse`, que implementa a interface `ObjetoGeometrico` (listagem 9.5) e que represente uma elipse com semi-eixos $a$ e $b$. Use como referência as classes `Circulo` e `Retangulo` (listagens 9.6 e 9.7). *Dica:* o perímetro de uma elipse é dado por $2\pi\sqrt{\frac{1}{2}(a^2+b^2)}$ (aproximadamente) e sua área por $\pi \times a \times b$. Veja também os exercícios 9.30 e 9.19.

**Exercício 13.162:** ⋆⋆

Na classe `Buffon` (listagem 13.7) usamos um array de três instâncias da classe `LinhaVerticalInfinita` para representar as linhas que podem ser interceptadas pelas agulhas. Um array foi usado para permitir a fácil extensão do programa para tratar de uma área maior para a simulação. Modifique o programa para que duas instâncias independentes da classe `LinhaVerticalInfinita` sejam usadas ao invés do array. Considerando o tempo de execução, existe alguma desvantagem em usarmos um array de instâncias se formos usar somente duas linhas verticais ?

**Exercício 13.163:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) o método `desvioMédio`, que retorna o desvio médio dos valores encapsulados no array. O desvio médio é calculado como $\frac{1}{N}\sum_{i=0}^{N-1}(x_i - \overline{x})$ onde $N$ é o número de elementos no array, $x_i$ é o elemento na posição $i$ e $\overline{x}$ é a média dos valores do array. *Dica:* valores infinitos, se houverem, não devem ser considerados, nem contados. Veja também o exercício 11.45.

**Exercício 13.164:** ⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) o método `médiaGeométrica`, que retorna a média geométrica dos valores encapsulados no array. A média geométrica é calculada como $\sqrt{x_0 \times x_1 \times x_2 \times x_3 \times \ldots \times x_{N-1}}$ onde $N$ é o número de elementos no array e $x_i$ é o elemento na posição $i$. *Dica:* valores infinitos, se houverem, não devem ser considerados. Veja também o exercício 11.45.

**Exercício 13.165:** ⋆⋆⋆

Crie na classe `ArrayDeFloats` (listagem 11.4) o método `desvioPadrão`, que retorna o desvio padrão dos valores encapsulados no array. O desvio padrão é calculado como

$$\sqrt{\frac{\sum_{i=0}^{N-1}(x_i - \overline{x})^2}{N-1}}$$

onde $N$ é o número de elementos no array, $x_i$ é o elemento na posição $i$ e $\overline{x}$ é a média dos valores do array. *Dica:* valores infinitos, se houverem, não devem ser considerados, nem contados. Veja também o exercício 11.45.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

**Exercício 13.166:** ★★★

Crie na classe `ArrayDeFloats` (listagem 11.4) o método `raizMédiaDosQuadrados`, que retorna a raiz da média dos quadrados dos valores encapsulados no array, que é calculada como

$$\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(x_i^2)}$$

onde $N$ é o número de elementos no array e $x_i$ é o elemento na posição $i$. *Dica:* valores infinitos, se houverem, não devem ser considerados, nem contados. Veja também o exercício 13.165.

**Exercício 13.167:** ★★★

Escreva a classe `RoboDeRotacaoLivre` que herda da classe `RoboAbstrato` (listagem 9.1) e que pode se movimentar livremente em qualquer ângulo. *Dica:* as novas coordenadas $(x_2, y_2)$ de um robô que se encontra nas coordenadas $(x_1, y_1)$ e orientado para o ângulo $\alpha$ opdem ser calculadas convertendo-se as coordenadas $(x_1, y_1)$ para coordenadas polares $(r, \alpha)$, adicionando-se a distância a ser percorrida ao valor $r$ e convertendo-se as coordenadas polares $(novor, \alpha)$ para as coordenadas retangulares $(x_2, y_2)$ (veja a figura 13.6). Use a classe `Ponto2D` na listagem 13.1 como referência. Opcionalmente, considere usar a classe desenvolvida no exercício 13.46.

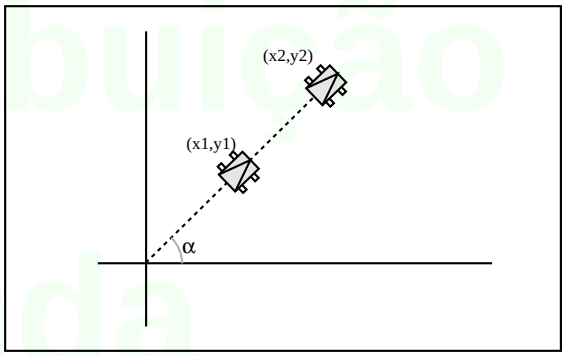



Figura 13.6: Movimento de um robô com orientação livre.

**Exercício 13.168:** ★★★★

A variância dos valores em um array pode calculada como sendo o quadrado do desvio padrão (veja exercício 13.165), ou

$$\frac{\sum_{i=0}^{N-1}(x_i - \bar{x})^2}{N-1}$$

(sendo mais eficiente calcular o desvio padrão a partir da variância do que o inverso). A mesma equação pode ser escrita como

$$\frac{1}{N-1}\left(\sum_{i=0}^{N-1}(x_i^2) - N\bar{x}^2\right)$$

Escreva dois métodos na classe `ArrayDeFloats` que calculem a variância, usando as duas equações mostradas. Escreva também uma aplicação que verifique se existe alguma vantagem em relação ao tempo de execução usando as duas diferentes equações para cálculo da variância, e explique os resultados.

**Exercício 13.169:** ★★★★★

O *casco convexo* de um conjunto de pontos é o subconjunto de pontos que delimita todos os outros. A figura 13.7 demonstra este conceito. O algoritmo de determinação do casco convexo é o seguinte:

1. Localize o ponto cuja coordenada *y* é a menor do conjunto. Se houverem dois ou mais pontos nestas condições, selecione o que tiver a maior coordenada *x*.
2. Calcule o ângulo entre este ponto de referência e todos os outros pontos do conjunto, ordenando do menor ângulo para o maior.
3. Selecione o ponto cujo ângulo formado com o ponto de referência seja o menor. Se este ponto for igual ao primeiro ponto, interrompa o algoritmo. O casco convexo será a lista de pontos obtidos até agora.
4. Se o ponto cujo ângulo formado com o ponto de referência não for o primeiro ponto, armazene este ponto como parte do casco convexo, transformando-o no ponto de referência atual e continue o algoritmo do passo 2.

Escreva para a classe `ArrayDePontos2D` (veja o exercício 11.98) um método `cascoConvexo` que retorne uma instância da própria classe contendo os pontos que compõem o casco convexo do conjunto de pontos encapsulado.

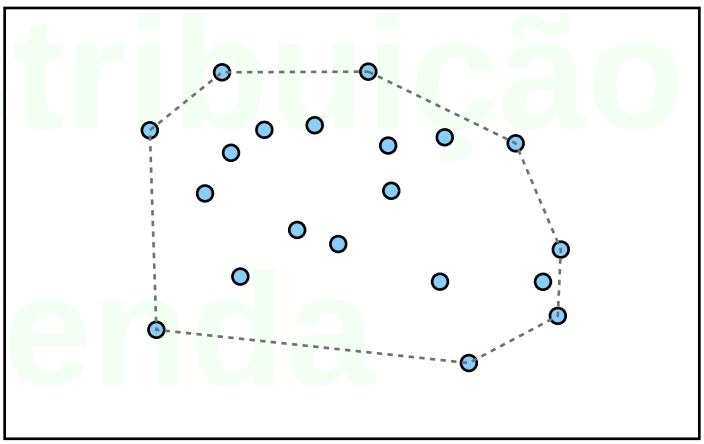


Figura 13.7: O casco convexo de um conjunto de pontos.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

# Parte III

# Classes de Java para Programas com Interfaces Gráficas

# Parte IV

# Projetos Sugeridos

# Parte V

# Apêndices

# Apêndice A

# Tipos de Dados em Java

## A.1 Introdução

Este apêndice apresenta os tipos de dados nativos de Java, com exemplos demonstrando usos corretos, incorretos e cujos resultados podem ser inesperados, assim como as classes-envelope de Java, que servem para representar valores de tipos nativos encapsulados em classes.

## A.2 Tipos nativos

Nesta seção veremos os tipos nativos (isto é, que são representados e manipulados sem a necessidade de uso de classes) de Java.

### A.2.1 O tipo nativo `boolean`

`boolean`

Campos e variáveis do tipo nativo `boolean` podem representar valores booleanos (verdadeiros ou falsos). Java define duas constantes, `true` e `false`, que representam os valores verdadeiro e falso, respectivamente. Ao contrário de outras linguagens como C e C++, não existem valores numéricos associáveis com `true` e `false` - operações envolvendo valores booleanos devem ser feitos exclusivamente com estas constantes ou operações que retornem um valor booleano.

Alguns exemplos de uso de variáveis do tipo `double` são mostrados no programa na listagem A.1.

Listagem A.1: A classe `DemoTipoBoolean`, que demonstra usos do tipo nativo `boolean`.

```
1  /**
2   * A classe DemoTipoBoolean, que demonstra vários usos corretos do tipo nativo boolean.
3   */
4  class DemoTipoBoolean
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo booleano, que receberão expressões booleanas diversas.
9    * @param argumentos os argumentos que podem ser passados para o método via linha
10   *        de comando, mas que neste caso serão ignorados.
11   */
12   public static void main(String[] argumentos)
13     {
14     boolean b1,b2,b3;               // declaração das variáveis
15     b1 = true;                      // b1 recebe a constante true
16     b2 = (1>2);                     // valor lógico de uma expressão (false)
17     b3 = (2+3 == 6);                // comparação de expressões (false)
18     boolean b4 = (1<2) && (3<4);    // "E" lógico de duas expressões (true)
19     boolean b5 = (1<2) || (3<2);    // "OU" lógico de duas expressões (true)
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
20     boolean b6 = !(false);          // "NÃO" lógico de uma constante (true)
21     boolean b7 = b1 && (b6 || b2);  // combinação de expressões (true)
22     boolean b8 = !(b1 && b6) || b2; // combinação de expressões (false)
23     }  // fim do método main
24
25   } // fim da classe DemoTipoBoolean
```

Na listagem A.1 vemos que podemos usar variáveis do tipo `boolean` para receber constantes (linha 15) e resultados de expressões de comparação, que são booleanos por natureza (linhas 16 a 19). Para operações com valores booleanos, usamos o sinal `&&` para representar a operação booleana **E**, o sinal `||` para representar a operação booleana **OU** e o sinal `!` para representar a operação booleana **NÃO**. Alguns exemplos de uso incorreto de campos do tipo `boolean` são mostrados no programa na listagem A.2.

Listagem A.2: A classe `DemoTipoBooleanComErros`, que demonstra usos incorretos do tipo nativo `boolean`.

```
1  /**
2   * A classe DemoTipoBooleanComErros, que demonstra vários usos incorretos do tipo
3   * nativo boolean.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoTipoBooleanComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo booleano, que receberão expressões que não são booleanas.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     boolean b1,b2,b3;             // declaração das variáveis
17     b1 = 0;                       // ERRO: tipo int diferente de boolean, não pode
18                                   //       ser convertido
19     b2 = (boolean)0;              // ERRO: cast de int para boolean inválido
20     b3 = FALSE;                   // ERRO: constante não definida
21     boolean b4 = 'F';             // ERRO: tipo char diferente de boolean, não pode
22                                   //       ser convertido
23     boolean b5 = "FALSE";         // ERRO: classe String diferente de boolean, não pode
24                                   //       ser convertida
25     boolean b6 = (boolean)"F";    // ERRO: cast de string para boolean inválido
26     }  // fim do método main
27
28   } // fim da classe DemoTipoBooleanComErros
```

No programa na listagem A.2, vemos que variáveis do tipo `boolean` não podem receber valores de tipos inteiros ou de ponto flutuante, mesmo com *cast* (linhas 17 e 19). As constantes `true` e `false` devem ser escritas com todas as letras minúsculas: variações não são reconhecidas (linha 20). O uso de constantes do tipo `char` ou da classe `String` também não são aceitas, nem mesmo com o *cast* (linhas 23 e 25).

### A.2.2 O tipo nativo `char`

O tipo `char` representa um único caracter em Java. Caracteres podem ser processados como valores numéricos que são mapeados para letras, dígitos, símbolos e outros, ou em conjunto como instâncias da classe `String` (em Java, strings não são arrays de caracteres). Caracteres em Java são representados usando o conjunto de caracteres Unicode, usando dois bytes por caracter. Esta representação de caracteres permite a representação de caracteres em várias línguas simultaneamente, não somente como caracteres a serem lidos, processados ou impressos, mas como parte

`char`

*Unicode*

dos próprios programas em Java[1]. O suporte para línguas estrangeiras é dado pelo sistema operacional - nem todos os caracteres poderão ser usados em qualquer sistema. Mais informações sobre Unicode podem ser encontradas no site http://www.unicode.org.

O tipo `char` é equivalente a um inteiro de 16 bits sem sinal, capaz de representar valores na faixa 0 a 65535, que podem até mesmo ser usados para operações simples. O programa na listagem A.3 mostra alguns exemplos válidos de uso do tipo `char`, onde vemos caracteres especificados diretamente entre aspas simples (’), por valores numéricos inteiros e até mesmo por operações matemáticas simples cujos resultados estejam na faixa de valores aceitáveis para `char`s:

Listagem A.3: A classe `DemoTipoChar`, que demonstra usos do tipo nativo `char`.

```
1  /**
2   * A classe DemoTipoChar, que demonstra vários usos corretos do tipo nativo char.
3   */
4  class DemoTipoChar
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo char, que receberão constantes e expressões cujo
9    * resultado é do tipo char.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     char c1,c2,c3,c4;    // declaração das variáveis
16     c1 = 'a';            // caracter ASCII normal
17     c2 = '!';            // caracter ASCII normal (símbolo)
18     c3 = 'ã';            // caracter especial (português)
19     c4 = 'ß';            // caracter especial (alemão)
20     char c5 = 'a'+1;     // operação válida: resultado 'b'
21     char c6 = 'B'-1;     // operação válida: resultado 'A'
22     char c7 = 123;       // valor Unicode especificado diretamente: caracter '{'
23     char c8 = 7;         // valor Unicode especificado diretamente: sinal sonoro
24     char c9 = '7';       // caracter Unicode '7' - diferente do sinal sonoro
25     char c10 = 1111/17; // caracter Unicode 65 ('A') - até mesmo esta operação é válida.
26     }  // fim do método main
27
28   } // fim da classe DemoTipoChar
```

Alguns caracteres são considerados especiais - se tentarmos imprimir estes caracteres, ao invés de um caracter normal uma ação especial ocorrerá. Um exemplo de caracter especial é o mostrado na linha 23 da listagem A.3: em alguns terminais, um caracter em branco será impresso e um som será emitido pelo computador - o caracter cujo valor é o número 7 corresponde ao caracter *bell*.

Outros exemplos de caracteres de controle são mostrados na tabela A.1. Alguns destes caracteres não podem ser entrados diretamente via teclado (pois não existem teclas correspondentes), e devem ser especificados com uma barra (\) seguida de um caracter comum.

Alguns exemplos de uso incorreto para valores do tipo `char` são mostrados no programa na listagem A.4.

[1] Caracteres não-alfanuméricos (acentos, letras estrangeiras, símbolos especiais, etc.) podem até aparecer em nomes de classes, métodos e variáveis, mas na prática deve-se evitar o uso de caracteres não-alfanuméricos para nomes de classes para evitar problemas com o sistema de arquivos dos computadores, que podem não reconhecer nomes de arquivos com estes caracteres.

| Caracter | Descrição |
|---|---|
| \" | Aspas duplas. Se for necessária a inclusão de aspas duplas em uma string, devemos usar esta composição pois aspas duplas são as delimitadoras de strings. Para especificação como caracteres a barra não é necessária, podemos escrever `char c = '"';`. |
| \' | Aspas simples. Se for necessária a criação de um caracter cujo conteúdo seja uma aspa simples, teremos que usar esta composição pois aspas simples são as delimitadoras de caracteres. Para especificação como parte de strings a barra não é necessária, podemos escrever `String nome = "O'Hara";`. |
| \n | Quebra de linha. Este caracter faz com que caracteres subsequentes em uma string sejam impressos em outra linha do terminal ou arquivo de texto. |
| \t | Tabulação. Este caracter força uma tabulação (espaçamento horizontal) em impressões subsequentes. |
| \\ | Esta combinação corresponderá à própria barra (\) em caracteres ou strings. |

Tabela A.1: Alguns caracteres de controle

Listagem A.4: A classe `DemoTipoCharComErros`, que demonstra usos incorretos do tipo nativo `char`.

```
1  /**
2   * A classe DemoTipoCharComErros, que demonstra vários usos incorretos do tipo
3   * nativo char.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoTipoCharComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo char, que receberão constantes e expressões que não são
11   * do tipo char.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     char c1,c2,c3;      // declaração das variáveis
18     c1 = 65536;         // ERRO: valor inteiro diferente de char, deve ser convertido
19                         //       para valor do tipo char com um cast
20     c2 = -1;            // ERRO: valor inteiro com sinal, deve ser convertido para valor
21                         //       do tipo char com um cast
22     c3 = 3.1416;        // ERRO: valor de ponto flutuante deve ser convertido para valor
23                         //       do tipo char com um cast, e precisão será perdida
24     char c4 = false;    // ERRO: constante do tipo boolean não pode ser convertida
25                         //       para char
26     char c5 = '';       // ERRO: caracter vazio não é constante válida do tipo char
27     char c6 = 'AB';     // ERRO: mais de um caracter entre aspas, constante inválida
28     char c7 = 'A'+'$'; // Não é um erro, mas resultado é 'e' (uma soma dos valores
29                         // 65 e 36 é efetuada, resultando em um outro caracter)
30     char c8 = "A";      // ERRO: instâncias da classe String não podem ser convertidas
31                         //       para caracteres. Valores do tipo char devem ser
32                         //       especificadas com aspas simples.
33     } // fim do método main
34
35   } // fim da classe DemoTipoCharComErros
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Alguns dos erros do programa na listagem A.4 podem ser evitados através do *cast* dos valores, mas mesmo assim perda de precisão pode ocorrer. O *cast* de um valor booleano para a variável `c4` não poderá ser feito, assim como o *cast* de uma string para um único caracter na linha 30. As linhas 26 e 27 apresentam erros de caracteres inválidos (constantes com menos ou mais de um único caracter). A atribuição feita à variável `c7`, embora não apresente erros de compilação ou execução, terá os valores somados ao invés de concatenados.

É importante lembrar que variáveis do tipo nativo `char` podem representar somente um caracter por vez. Para representação de vários caracteres (cadeia de caracteres ou strings), veja a seção 12.2.

### A.2.3 O tipo nativo `byte`

byte

Campos e variáveis do tipo `byte` podem representar valores inteiros na faixa entre $-128$ a 127, que podem ser atribuídos diretamente ou através de operações. Alguns exemplos de uso do tipo nativo `byte` são mostrados no programa na listagem A.5.

Listagem A.5: A classe `DemoTipoByte`, que demonstra usos do tipo nativo `byte`.

```
1  /**
2   * A classe DemoTipoByte, que demonstra vários usos corretos do tipo nativo byte.
3   */
4  class DemoTipoByte
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo byte, que receberão constantes e expressões cujo
9    * resultado é do tipo byte.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     byte b1,b2;          // declaração das variáveis
16     b1 = -1;             // b1 valerá -1
17     b2 = 123-2;          // b2 valerá 121
18     byte b3 = 122-123;   // b3 valerá -1
19     byte b4 = 12*10;     // b4 valerá 120
20     byte b5 = 12/10;     // b5 valerá 1 (divisão com valores inteiros)
21     }  // fim do método main
22
23   } // fim da classe DemoTipoByte
```

Na linha 20 da listagem A.5, somente a parte inteira do resultado foi considerada, já que o tipo nativo `byte` não pode representar casas decimais. A operação mostrada na linha 19 é válida (não gera erros de compilação nem de execução) porque o resultado estará na faixa de valores válidos para o tipo nativo `byte` e os valores da operação são constantes.

Alguns exemplos de uso incorreto do tipo nativo `byte` são mostrados no programa na listagem A.6, com comentários indicando os erros.

Listagem A.6: A classe `DemoTipoByteComErros`, que demonstra usos incorretos do tipo nativo `byte`.

```
1  /**
2   * A classe DemoTipoByteComErros, que demonstra vários usos incorretos do tipo
3   * nativo byte.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
5   */
6  class DemoTipoByteComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo byte, que receberão constantes e expressões que não são
11   * do tipo byte.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     byte b1,b2,b3;       // declaração das variáveis
18     b1 = 123456;         // ERRO: valor inteiro diferente de byte, deve ser convertido
19                          //       para valor do tipo byte com um cast
20     b2 = 1.24;           // ERRO: valor double diferente de byte, deve ser convertido
21                          //       para valor do tipo byte com um cast
22     b3 = 12000.24;       // ERRO: valor double diferente de byte, deve ser convertido
23                          //       para valor do tipo byte com um cast
24     byte b4 = true;      // ERRO: constante do tipo boolean não pode ser convertida
25                          //       para byte
26     byte b5 = 10;        // Não apresenta erro, mas...
27     byte b6 = b5*b5;     // ERRO: resultado da multiplicação é um inteiro, este
28                          //       resultado deve ser convertido para byte com um cast
29     } // fim do método main
30
31   } // fim da classe DemoTipoByteComErros
```

Na listagem A.6, vários erros ocorreram porque tentamos colocar valores incompatíveis com a capacidade das variáveis tipo `byte`. Variáveis declaradas como sendo do tipo `byte` não podem receber valores do tipo `boolean`, de qualquer tipo inteiro diferente de `byte` ou de qualquer tipo de ponto flutuante sem *cast*, mas mesmo que isto seja possível, poderemos ter perda de precisão ou de informação.

No exemplo de conversão da linha 18 da listagem A.6, o valor inteiro `123456` foi convertido para 64. Vamos examinar esta conversão com mais detalhes: um número inteiro é representado por 32 bits (veja seção A.2.5), mas uma variável do tipo `byte` somente usa 8 bits, então os 24 bits superiores são eliminados (descartados completamente), como mostrado a seguir:

| | | |
|---|---|---|
| `123456` em 32 bits: | `00000000000000011110001001000000` | |
| 24 bits superiores eliminados: | `........................01000000` | Resultado: `64` |

Para a conversão mostrada na linha 7 da listagem A.6, Java converterá primeiro o valor de ponto flutuante para um valor inteiro de 32 bits, e eliminará os 24 bits superiores. Detalhes da conversão são mostrados a seguir:

| | | |
|---|---|---|
| `12000` em 32 bits: | `00000000000000000010111011100000` | |
| 24 bits superiores eliminados: | `........................11100000` | Resultado: `-32` |

Neste exemplo de conversão, o primeiro bit do valor do tipo `byte` determina o sinal do valor, através da subtração de 128 do valor expresso pelos sete bits restantes.

### A.2.4 O tipo nativo `short`

`short`

Campos e variáveis do tipo `short` podem representar valores inteiros com 16 bits de precisão, na faixa -32768 a 32767. Variáveis e campos deste tipo podem receber valores diretamente ou através de operações, como mostrado no programa na listagem A.7.

Listagem A.7: A classe `DemoTipoShort`, que demonstra usos do tipo nativo `short`.

```
/**
 * A classe DemoTipoShort, que demonstra vários usos corretos do tipo nativo short.
 */
class DemoTipoShort
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações
  * de várias variáveis do tipo short, que receberão constantes e expressões cujo
  * resultado é do tipo short.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    short s1,s2,s3;      // declaração das variáveis
    s1 = 1000;           // s1 valerá 1000
    s2 = 1-1000;         // s2 valerá -999
    s3 = 100*100;        // s3 valerá 10000
    short s4 = 'A';      // s4 valerá 65 (valor do caracter 'A')
    short s5 = s3;       // s5 valerá 10000
    short s6 = 1999/10;  // s6 valerá 199 (divisão com valores inteiros)
    short s7 = -'A';     // estranho mas válido: s7 valerá -65.
    }  // fim do método main

  } // fim da classe DemoTipoShort
```

Os comentários da classe mostrada na listagem A.7 esclarecem alguns detalhes das atribuições e operações com valores do tipo `short`.

Alguns exemplos de uso incorreto de campos e variáveis do tipo `short` são mostrados no programa na listagem A.8.

Listagem A.8: A classe `DemoTipoShortComErros`, que demonstra usos incorretos do tipo nativo `short`.

```
/**
 * A classe DemoTipoShortComErros, que demonstra vários usos incorretos do tipo
 * nativo short.
 * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
 */
class DemoTipoShortComErros
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * várias variáveis do tipo short, que receberão constantes e expressões que não são
  * do tipo short.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    short s1,s2;         // declaração das variáveis
    s1 = 123456;         // ERRO: valor inteiro diferente de short, deve ser convertido
                         //       para valor do tipo short com um cast
    s2 = 100.1;          // ERRO: valor double diferente de short, deve ser convertido
                         //       para valor do tipo short com um cast
    short s3 = true;     // ERRO: constante do tipo boolean não pode ser convertida
                         //       para short
    short s4 = "A";      // ERRO: instância da classe String não pode ser convertida
                         //       para short, nem mesmo através do cast
    short s5 = 999*999;  // ERRO: resultado será do tipo inteiro, fora da faixa válida
                         //       para shorts, devendo ser convertido através de cast
    } // fim do método main
```

```
29
30   } // fim da classe DemoTipoShortComErros
```

Todos os erros acima serão indicados pelo compilador. Os erros das linhas 18, 20 e 26 poderão ser resolvidos através do *cast*, mas com perda de precisão ou informação.

### A.2.5 O tipo nativo `int`

Campos e variáveis do tipo nativo `int` podem representar valores numéricos inteiros que estejam dentro da faixa -2147483648 a 2147483647, usando 4 bytes para representá-los. Alguns exemplos de uso de variáveis do tipo `int` são mostrados no programa na listagem A.9. int

Listagem A.9: A classe `DemoTipoInt`, que demonstra usos do tipo nativo `int`.

```
1  /**
2   * A classe DemoTipoInt, que demonstra vários usos corretos do tipo nativo int.
3   */
4  class DemoTipoInt
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo int, que receberão constantes e expressões cujo
9    * resultado é do tipo int.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     int i1,i2;          // declaração das variáveis
16     i1 = -1000000;      // i1 valerá -1000000
17     i2 = -i1*2;         // i2 valerá 2000000
18     int i3 = i1+i2;     // i3 valerá 1000000
19     int i4 = 123*200;   // i4 valerá 24600
20     int i5 = 123/200;   // i5 valerá 0 (divisão com valores inteiros)
21     int i6,i7;          // declaração das variáveis
22     i6 = 123/120;       // i6 valerá 1 (divisão com valores inteiros)
23     i7 = 'X';           // i7 valerá 88 (valor do caracter 'X')
24     }  // fim do método main
25
26   } // fim da classe DemoTipoInt
```

Como mostrado na listagem A.9, variáveis e campos do tipo `int` podem receber constantes e valores resultantes de operações com inteiros.

Alguns exemplos de uso incorreto de valores do tipo `int` são mostrados no programa na listagem A.10.

Listagem A.10: A classe `DemoTipoIntComErros`, que demonstra usos incorretos do tipo nativo `int`.

```
1  /**
2   * A classe DemoTipoIntComErros, que demonstra vários usos incorretos do tipo
3   * nativo int.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoTipoIntComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo int, que receberão constantes e expressões que não são
11   * do tipo int.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     int i1,i2;          // declaração das variáveis
18     i1 = 99999*99999;   // ERRO: Nem o compilador nem a máquina virtual acusarão erros,
19                         //       mas o resultado será 1409865409 ao invés de 9999800001.
20                         //       Este tipo de erro é conhecido como overflow - o tipo
21                         //       int não pode representar o valor esperado então o
22                         //       trunca.
23     i2 = 0.00001;       // ERRO: valor do tipo double diferente de int, deve ser
24                         //       convertido para valor do tipo int através de cast.
25     int i3 = true;      // ERRO: constante do tipo boolean, não pode ser convertida para
26                         //       valor do tipo int, nem mesmo através de cast
27     int i4 = "X";       // ERRO: instância da classe String não pode ser convertida
28                         //       para int, nem mesmo através de cast.
29     } // fim do método main
30
31   } // fim da classe DemoTipoIntComErros
```

overflow

Um erro particularmente grave é mostrado na linha 18: nem o compilador nem o interpretador indicarão que a conta será calculada incorretamente. Este erro é chamado de *overflow*. Todos os outros erros serão indicados pelo compilador, e o da linha 23 pode ser resolvido através do *cast*, mas com perda de precisão ou informação.

### A.2.6 O tipo nativo `long`

long

Variáveis e campos do tipo nativo `long` podem representar valores numéricos inteiros que estejam dentro da faixa de valores -9223372036854775808 a 9223372036854775807, usando 8 bytes para representá-los. Alguns exemplos de uso de variáveis do tipo `long` são mostrados no programa na listagem A.11.

Listagem A.11: A classe `DemoTipoLong`, que demonstra usos do tipo nativo `long`.

```
1  /**
2   * A classe DemoTipoLong, que demonstra vários usos corretos do tipo nativo long.
3   */
4  class DemoTipoLong
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo long, que receberão constantes e expressões cujo
9    * resultado é do tipo long.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     long l1,l2,l3;            // declaração das variáveis
16     l1 = 9999999999L;         // l1 valerá 9999999999
17     l2 = 999999*999999L;      // l2 valerá 999998000001
18     l3 = 1/2;                 // l3 valerá 0 (divisão com valores inteiros)
19     long l4,l5;               // declaração das variáveis
20     l4 = 3/2;                 // l4 valerá 1 (divisão com valores inteiros)
21     l5 = 'A'*'B'*'C';         // l5 valerá 287430 (multiplicação dos valores 65, 66
22                               // e 67)
23     }  // fim do método main
24
25   } // fim da classe DemoTipoLong
```

Como demonstrado nas linhas 16 e 17 da listagem A.11 devemos especificar valores constantes do tipo `long` com a letra `L` como sufixo. Este `'L'` pode ser maiúsculo ou minúsculo, mas como o `'l'` minúsculo assemelha-se ao dígito `'1'`, é preferível o uso da forma maiúscula. Alguns

exemplos de uso incorreto de campos e variáveis do tipo `long` são mostrados no programa na listagem A.12.

Listagem A.12: A classe `DemoTipoLongComErros`, que demonstra usos incorretos do tipo nativo `long`.

```
1  /**
2   * A classe DemoTipoLongComErros, que demonstra vários usos incorretos do tipo
3   * nativo long.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoTipoLongComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo long, que receberão constantes e expressões que não são
11   * do tipo long.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     long l1,l2,l3,l4,l5,l6; // declaração das variáveis
18     l1 =       9999999999;  // ERRO: compilador considera o valor como inteiro, mesmo
19                             //       que seja passado para uma variável do tipo long.
20                             //       Como o valor é muito grande para ser representado
21                             //       por um inteiro, ocorre erro de compilação.
22     l2 = (long)9999999999;  // ERRO: mesma razão do erro acima. O compilador considera
23                             //       o valor como sendo inteiro mesmo com a conversão
24                             //       explícita.
25     l3 = 3.1415926536;      // ERRO: valor do tipo double diferente de long, deve ser
26                             //       convertido para valor do tipo long através de
27                             //       cast.
28     l4 = 10000*10000.1;     // ERRO: um dos valores é do tipo double, cast é
29                             //       necessário.
30     l5 = 9999999999999999L; // 15 valerá 9999999999999999
31     l6 = 15*15;             // ERRO: Nem o compilador nem a máquina virtual acusarão
32                             //       erros, mas o resultado será -8834407033341083647
33                             //       ao invés de 99999999999999980000000000000001.
34                             //       Este tipo de erro é conhecido como overflow - o
35                             //       tipo long não pode representar o valor esperado
36                             //       então o trunca.
37     long l7,l8;             // declaração das variáveis
38     l7 = true;              // ERRO: constante do tipo boolean, não pode ser convertida
39                             //       para valor do tipo long, nem mesmo através de cast
40     l8 = "1234567890";      // ERRO: instância da classe String não pode ser convertida
41                             //       para long, nem mesmo através de cast.
42     } // fim do método main
43
44   } // fim da classe DemoTipoLongComErros
```

A maioria dos erros na listagem A.12 já foram vistos em exemplos de outros tipos de dados. O erro mais crítico é o de *overflow*, que ocorre na linha 31. Vale a pena lembrar que este tipo de erro não é registrado em tempo de compilação ou de execução, e pode ser difícil de ser identificado em um programa.

### A.2.7 O tipo nativo `float`

Campos e variáveis do tipo nativo `float` podem representar valores numéricos com ponto flutuante que estejam dentro da faixa 1.40129846432481707e-45 a 3.40282346638528860e+38 (positivos e negativos)[2] mas sem precisão exata. O tipo nativo `float` usa a mesma memória do tipo

`float`

[2] A notação `xey` onde `x` e `y` são valores numéricos é usada para representar alguns valores de ponto flutuante: `xey` corresponde a $x \times 10^y$.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

`int` (4 bytes). Alguns exemplos de uso de variáveis do tipo `float` são mostrados no programa na listagem A.13.

Listagem A.13: A classe `DemoTipoFloat`, que demonstra usos do tipo nativo `float`.

```
1  /**
2   * A classe DemoTipoFloat, que demonstra vários usos corretos do tipo nativo float.
3   */
4  class DemoTipoFloat
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo float, que receberão constantes e expressões cujo
9    * resultado é do tipo float.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     float f1,f2,f3,f4,f5,f6;    // declaração das variáveis
16     f1 = (float)3.1415926536;   // f1 valerá 3.1415927 - note o arredondamento
17     f2 = 3.14159265358979F;     // f2 valerá 3.1415927 - note o arredondamento
18     f3 = f1-f2;                 // f3 valerá 0
19     f4 = 1/2;                   // f4 valerá 0 ! (divisão de inteiros)
20     f5 = 1F/2;                  // f5 valerá 0.5
21     f6 = 1/2F;                  // f6 valerá 0.5
22     float f7,f8,f9,f10,f11;     // declaração das variáveis
23     f7 = 'A'*'B'*'C';           // f7 valerá 287430.0 (multiplicação dos
24                                 // valores 65,66 e 67)
25     f8 = 1F/'2';                // f8 valerá 0.02 (divide 1.0 pelo caracter '2'
26                                 // que vale 50)
27     f9 = 1f/0f;                 // f9 valerá Infinity
28     f10 = -1f/0f;               // f10 valerá -infinity
29     f11 = 0f/0f;                // f11 valerá NaN
30     }  // fim do método main
31
32   } // fim da classe DemoTipoFloat
```

Para usar constantes do tipo `float` em programas, devemos colocar o sufixo `'F'` (maiúsculo ou minúsculo) após as constantes, como mostrado na linhas 17, 20, 21, 25 e 27 a 29 da listagem A.13. Os valores representados por um `float` não são representados exatamente, como pode ser visto nos comentários das linhas 16 e 17.

Algumas regras relativas às constantes também são válidas para o tipo `float`: constantes numéricas sem o sufixo `'F'` nem pontos flutuantes são interpretadas como sendo do tipo `int` e tratadas como tal: a divisão mostrada na linha 19 é considerada uma divisão de inteiros, e seu resultado é zero. Nas linhas 20 e 21, como uma das constantes na divisão é de ponto flutuante (por causa do sufixo `'F'`), o resultado é considerado como ponto flutuante do tipo `float`.

*infinito*

`NaN` (Not a Number)

Três atribuições curiosas foram feitas na listagem A.13: a linguagem Java permite a representação e até mesmo processamento dos valores infinito (resultado da divisão de qualquer valor por zero) e `NaN` (*Not a Number*), resultado de algumas operações indefinidas. Algumas regras especiais, mostradas a seguir, ditam como estes valores se comportarão em operações aritméticas, mas em geral os resultados seguem as regras matemáticas.

Valores infinitos podem ser obtidos através da divisão de qualquer valor por zero. Dependendo dos sinais dos operandos, o valor infinito será positivo ou negativo. Caso o resultado de uma operação seja maior do que o maior valor que puder ser representado por um `float`, este resultado também será infinito (positivo ou negativo). A multiplicação de qualquer valor não-zero por

infinito também dará como resultado infinito, e a divisão de infinito por qualquer valor não-zero será infinita.

Valores `NaN` serão resultado de qualquer expressão que envolva um valor `NaN`. Este resultado também ocorrerá quando dividirmos infinito por infinito ou tentarmos subtrair infinito de infinito. A multiplicação de zero por infinito e a divisão de zero por zero também resultarão em `NaN`. Apesar das operações $0^0$ e $0^\infty$ serem consideradas matematicamente indefinidas, Java as considera como iguais a 1. Mais exemplos e informações sobre valores numéricos especiais podem ser encontradas na seção 13.2.

Alguns exemplos de uso incorreto de valores do tipo `float` ou usos que podem dar resultados inesperados são mostrados no programa na listagem A.14.

Listagem A.14: A classe `DemoTipoFloatComErros`, que demonstra usos incorretos do tipo nativo `float`.

```
1  /**
2   * A classe DemoTipoFloatComErros, que demonstra vários usos incorretos do tipo
3   * nativo float.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoTipoFloatComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo float, que receberão constantes e expressões que não são
11   * do tipo float.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     float f1,f2,f3,f4,f5;             // declaração das variáveis
18     f1 = 1234567890;                  // f1 valerá 1.23456794E9 - note o arredondamento
19     f2 = 123456.0000000001F;          // f2 valerá 123456.0 - note o arredondamento
20     f3 = (1f/7)+(1f/7)+(1f/7)-(3f/7); // f3 valerá 2.9802322E-8 - note o arredondamento
21     f4 = false;                       // ERRO: constante do tipo boolean, não pode ser
22                                       //       convertida  para valor do tipo float,
23                                       //       nem mesmo através de cast
24     f5 = "3.1416";                    // ERRO: instância da classe String não pode ser
25                                       //       convertida para valor do tipo float,
26                                       //       nem mesmo através de cast
27     float f6 = 1.0;                   // ERRO: valor 1.0 é implicitamente do tipo
28                                       //       double, precisa ser convertido
29                                       //       explicitamente para float para poder
30                                       //       ser atribuído
31     float f7 = 1/0;                   // ERRO: a operação 1/0 será feita com valores
32                                       //       do tipo int antes de ser armazenada em
33                                       //       um float: divisão de inteiros por zero
34                                       //       causa exceção.
35     } // fim do método main
36
37   } // fim da classe DemoTipoFloatComErros
```

No programa na listagem A.14, temos mais problemas de arredondamento devidos à capacidade finita do tipo `float` de representar dados: nas linhas 18 e 19 vemos o arredondamento dos valores, porque o tipo `float` é incapaz de armazenar grandes valores com detalhes. Vale a pena notar que estes "detalhes" podem fazer muita diferença - imagine usar um `float` para armazenar valores monetários e descobrir que muito dinheiro está sumindo por causa de erros de arredondamento.

Outro erro de arredondamento ocorre na linha 20: a soma das frações mostradas deveria ser exatamente zero, mas por causa do arredondamento das frações, o resultado é um valor muito pequeno mas diferente de zero. Este erro pode causar problemas maiores em um programa caso façamos comparações entre pontos flutuantes: se uma comparação entre `(1f/7)+(1f/7)+(1f/7)` e `(3f/7)` fosse feita, o resultado ia ser `false`.

Nas linhas 21 e 24 da listagem A.14 vemos que o tipo `float` sofre das mesmas restrições dos outros tipos de dados em relação aos tipos de variáveis que podem ser convertidas através do *cast* para `float`.

### A.2.8 O tipo nativo `double`

`double` Campos e variáveis do tipo nativo `double` podem representar valores numéricos com ponto flutuante que estejam dentro da faixa 4.94065645841246544e-324 a 1.79769313486231570e+308 (positivos e negativos) com mais precisão numérica do que o tipo `float`. O tipo nativo `double` usa a mesma memória do tipo `long` (8 bytes). O tipo `double` também pode manipular valores infinitos e `NaN`, com as mesmas regras vistas para o tipo `float`. Veja a seção 13.2 para mais informações sobre valores numéricos especiais.

Alguns exemplos de uso de variáveis do tipo `double` são mostrados no programa na listagem A.15.

Listagem A.15: A classe `DemoTipoDouble`, que demonstra usos do tipo nativo `double`.

```
1  /**
2   * A classe DemoTipoDouble, que demonstra vários usos corretos do tipo nativo double.
3   */
4  class DemoTipoDouble
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias variáveis do tipo double, que receberão constantes e expressões cujo
9    * resultado é do tipo double.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     double d1,d2,d3,d4,d5,d6;  // declaração das variáveis
16     d1 = 1.2345678901;         // d1 valerá 1.2345678901 - armazenado exato
17     d2 = 1.234567890123456789; // d2 valerá 1.2345678901234567 - armazenado arredondado
18     d3 = d1+999999;            // d3 valerá 1000000.2345678901 - armazenado arredondado
19     d4 = 1/2;                  // d4 valerá 0! : divisão de inteiros
20     d5 = 1./2;                 // d5 valerá 0.5
21     d6 = 1/2d;                 // d6 valerá 0.5
22     double d7,d8,d9;           // declaração das variáveis
23     d7 = 1./'2';               // d7 valerá 0.02 (divide 1.0 pelo caracter '2' cujo
24                                //    valor inteiro é 50)
25     d8 = 1e72;                 // d8 valerá 1x10^72 ou 1E72
26     d9 = 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000.;
27                                // d9 também valerá 1x10^72 ou 1E72
28     }  // fim do método main
29
30   } // fim da classe DemoTipoDouble
```

No programa na listagem A.15 vemos que o tipo `double` pode representar valores com maior precisão do que o tipo `float`, apesar de em alguns casos ocorrer o arredondamento, como nas



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

linhas 17 e 18.

Constantes do tipo `double` podem ser escritas usando o sufixo `'D'` (maiúsculo ou minúsculo) ou um ponto flutuante (`.`). O ponto flutuante pode ser usado mesmo em constantes sem casas decimais, como mostrado nas linhas 20, 21 e 23. Constantes com valores muito grandes ou pequenos podem ser escritas com a notação `xey` ou diretamente, como mostrado nas linhas 25 e 26. Alguns exemplos de uso incorreto de campos e variáveis do tipo `double` ou usos que podem dar resultados inesperados são mostrados no programa na listagem A.16.

Listagem A.16: A classe `DemoTipoDoubleComErros`, que demonstra usos incorretos do tipo nativo `double`.

```
1  /**
2   * A classe DemoTipoDoubleComErros, que demonstra vários usos incorretos do tipo
3   * nativo double.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoTipoDoubleComErros
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias variáveis do tipo double, que receberão constantes e expressões que não são
11   * do tipo double.
12   * @param argumentos os argumentos que podem ser passados para o método via linha
13   *        de comando, mas que neste caso serão ignorados.
14   */
15   public static void main(String[] argumentos)
16     {
17     double d1,d2,d3,d4;          // declaração das variáveis
18     d1 = 1000000.000000000001;   // d1 valerá 1000000.0 - note o arredondamento
19     d2 = (1/987654.)+(2/987654.)-(3/987654.);
20                                  // d2 valerá 4.2351647362715017E-22 - note o
21                                  // arredondamento
22     d3 = false;                  // ERRO: constante do tipo boolean, não pode ser
23                                  //       convertida  para valor do tipo double, nem
24                                  //       mesmo através de cast
25     d4 = "3.1416";               // ERRO: instância da classe String não pode ser
26                                  //       convertida para valor do tipo double, nem
27                                  //       mesmo através de cast
28     double d5 = (byte)2.5;       // d5 valerá 2.0 (o cast para byte eliminou as casas
29                                  // decimais, mas na atribuição o valor voltou a ser
30                                  // de ponto flutuante
31     } // fim do método main
32
33   } // fim da classe DemoTipoDoubleComErros
```

No programa na listagem A.16 temos mais erros de arredondamento, nas linhas 18 e 19, e erros de tipos que não podem ser convertidos, nas linhas 22 e 25.

## A.3 Classes que encapsulam valores nativos

Nesta seção veremos classes que correspondem aos tipos nativos vistos na seção anterior. Estas classes possibilitam a representação de valores nativos como classes, o que é particularmente útil para uso em métodos que esperam um argumento que seja um herdeiro da classe `Object`. As classes, entretanto, não permitem a manipulação direta do conteúdo encapsulado: com estas classes não é possível efetuar as operações que podemos fazer com tipos nativos.

Todas as classes mostradas nesta seção fazem parte do pacote `java.lang`, então não é necessário nenhum comando `import` para utilizá-las.

### A.3.1 A classe `Boolean`

*classe* `Boolean`

A classe `Boolean` possibilita o encapsulamento de valores do tipo nativo `boolean`. Os construtores da classe permitem a inicialização do valor encapsulado através da especificação de um valor do tipo `boolean` ou de uma string que contenha um valor do tipo `boolean`. Neste segundo caso, se a string for igual a "`true`" (independente de estar em caracteres maiúsculos, minúsculos ou misturados) o valor representado será igual a `true`, caso contrário será igual a `false`. Não existem mecanismos nesta classe para conversão de ou para valores numéricos, já que o tipo `boolean` não é compatível com valores numéricos.

Alguns campos e métodos interessantes e úteis desta classe são:

*campo* `TRUE`
- O campo estático `TRUE` é uma instância da classe `Boolean` que encapsula o valor booleano `true`.

*campo* `FALSE`
- O campo estático `FALSE` é uma instância da classe `Boolean` que encapsula o valor booleano `false`.

*método* `valueOf`
- O método `valueOf` é uma fábrica de instâncias que aceita como argumento uma string e cria uma nova instância da classe `Boolean` encapsulando o valor `true` se a string passada como argumento for igual (independentemente de maiúsculas ou minúsculas) a "`true`". Se a string for diferente de "`true`", cria uma instância que encapsula o valor booleano `false`.

*método* `booleanValue`
- O método `booleanValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `boolean`. Este método é o método usado para "desencapsular" o valor booleano encapsulado pela classe (é importante lembrar que não podemos usar instâncias da classe `Boolean` como argumentos para comandos `if`).

*método* `equals`
- O método `equals` aceita como argumento uma instância de qualquer classe mas retorna `true` somente se esta instância for da classe `Boolean` e encapsular o mesmo valor que é encapsulado pela instância que executa o método.

*método* `toString`
- O método `toString` retorna o valor encapsulado como uma instância da classe `String`. Este método pode ser chamado implicitamente como os outros métodos `toString` de outras classes.

Alguns exemplos de uso de instâncias da classe `Boolean` são mostrados no programa na listagem A.17.

Listagem A.17: A classe `DemoClasseBoolean`, que demonstra usos da classe `Boolean`.

```
1  /**
2   * A classe DemoClasseBoolean, que demonstra vários usos corretos da classe Boolean.
3   */
4  class DemoClasseBoolean // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações de
8    * várias instâncias da classe Boolean, que encapsula um valor do tipo nativo boolean.
9    * O método também demonstra algumas operações com estas instâncias.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     Boolean b1,b2,b3,b4;         // declaração das instâncias
16     b1 = new Boolean(true);      // b1 representará o Boolean true.
17     b2 = new Boolean("true");    // b2 representará o Boolean true. A conversão de
18                                  // string para Boolean será feita automaticamente.
19     b3 = new Boolean('a'=='a'); // b3 representará o Boolean true. A expressão entre
```

```
20                                  // parênteses é avaliada como sendo verdadeira, e seu
21                                  // resultado será passado para o construtor da classe
22                                  // Boolean.
23      b4 = new Boolean("t");      // b4 representará o Boolean false. Veja no texto a
24                                  // descrição das regras de conversão.
25      boolean bl1 = b1.booleanValue(); // recupera o valor encapsulado por b1 como um
26                                       // valor do tipo boolean
27      boolean bl3 = b3.booleanValue(); // recupera o valor encapsulado por b3 como um
28                                       // valor do tipo boolean
29      // Compara dois Booleans. Note que não podemos comparar estes valores diretamente
30      // com == e !=.
31      if (b1.equals(b2)) System.out.println(b1+" é igual a "+b2);
32      if (!b3.equals(b4)) System.out.println(b3+" é diferente de "+b4); // note o !
33      Boolean b5 = Boolean.valueOf("TRUE");  // b5 representará o Boolean true,
34                                             // convertido da string que representa este
35                                             // valor booleano.
36      }
37
38    } // fim da classe DemoClasseBoolean
```

Nas linhas 16 a 23 da listagem A.17 temos exemplos dos construtores da classe `Boolean`, o primeiro e terceiro recebendo um valor do tipo `boolean` como argumento e o segundo e quarto recebendo strings como argumentos. Nos três primeiros exemplos, o valor encapsulado será igual a `true`, enquanto que no quarto será igual a `false` pois a string passada como argumento para o construtor é diferente de "`true`". Nas linhas 25 e 27 convertemos o valor contido em instâncias da classe para variáveis do tipo booleano.

Alguns exemplos de uso incorreto de instâncias desta classe são mostrados no programa na listagem A.18.

Listagem A.18: A classe `DemoClasseBooleanComErros`, que demonstra usos incorretos da classe `Boolean`.

```
1  /**
2   * A classe DemoClasseBooleanComErros, que demonstra vários usos incorretos e algumas
3   * armadilhas da classe Boolean.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoClasseBooleanComErros // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias instâncias da classe Boolean, que encapsula um valor do tipo nativo boolean.
11   * O método também demonstra algumas operações com estas instâncias. Muitas das
12   * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
13   * compilado.
14   * @param argumentos os argumentos que podem ser passados para o método via linha
15   *        de comando, mas que neste caso serão ignorados.
16   */
17    public static void main(String[] argumentos)
18      {
19      Boolean b1,b2,b3,b4;        // declaração das instâncias
20      b1 = false;                 // ERRO: valor do tipo nativo boolean não pode ser
21                                  //       convertido para instância da classe Boolean.
22      b2 = new Boolean(0);        // ERRO: valor do tipo nativo int não pode ser
23                                  //       convertido para instância da classe Boolean.
24      b3 = new Boolean();         // ERRO: o construtor da classe Boolean espera um
25                                  //       argumento.
26      b4 = new Boolean(" true"); // Não existe erro, mas b4 encapsulará o valor booleano
27                                  // false pois o argumento é diferente de "true".
28      // Compara Booleans com instâncias de outras classes, com o valor null e com
29      // valores de tipos nativos.
30      if (b4.equals(null)) System.out.println(b4+" é nulo."); // nada será impresso
31      if (b1.equals(true)) System.out.println(b1+" é true."); // ERRO: é impossível
```

```
32                                                          // comparar argumentos que
33                                                          // não sejam instâncias
34                                                          // com o método equals.
35      if (b1.equals(1)) System.out.println(b1+" vale um."); // o mesmo erro do caso
36                                                          // acima.
37      Character c = new Character('t'); // criamos uma instância da classe Character
38      if (b1.equals(c))                 // nada será impresso, pois embora b1 e c
39        System.out.println(b1+" é igual a "+c); // sejam instâncias, são instâncias de
40                                                // classes diferentes, não podendo ser
41                                                // comparadas - o resultado será sempre
42                                                // false.
43    }
44
45  } // fim da classe DemoClasseBooleanComErros
```

Nas linhas 20, 22 e 24 da listagem A.18 temos três criações incorretas de instâncias da classe `Boolean`, na primeira tentando usar diretamente um valor do tipo nativo `boolean`, na segunda passando um valor inteiro e na terceira, nenhum valor ao construtor. O resto da listagem mostra erros de comparação de uma instância da classe `Boolean` com instâncias de outras classes.

### A.3.2 A classe `Character`

*classe* `Character`

A classe `Character` é uma classe de Java que encapsula valores do tipo nativo `char`. O construtor único da classe permite a inicialização do valor encapsulado através da especificação de um valor do tipo `char` como argumento. A classe `Character` tem, além de métodos de conversão do valor encapsulado para outros tipos nativos, métodos estáticos para conversão entre caracteres maiúsculos e minúsculos, métodos para verificação do tipo do caracter (espaço, dígito, letra, etc.) e métodos para auxílio na conversão de bases.

Alguns dos campos e métodos mais úteis da classe `Character` são:

*campo* `MIN_VALUE`
- O campo estático `MIN_VALUE` do tipo nativo `char` é igual ao menor valor possível para um campo ou variável do tipo `char`, sendo igual a 0.

*campo* `MAX_VALUE`
- O campo estático `MAX_VALUE` do tipo nativo `char` é igual ao maior valor possível para um campo ou variável do tipo `char`, sendo igual a 65535.

*campo* `MIN_RADIX`
- O campo estático `MIN_RADIX` (do tipo `int`) é igual a menor base que pode ser usada para conversão de bases, sendo igual a 2 (ou seja, a base mínima para representação e conversão é a binária).

*campo* `MAX_RADIX`
- O campo estático `MAX_RADIX` (do tipo `int`) é igual a maior base que pode ser usada para conversão de bases, sendo igual a 36. Esta base em particular usa todos os dígitos numéricos mais todas as letras do alfabeto para representação e conversão.

*método* `charValue`
- O método `charValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo `char`, sendo o método usado para "desencapsular" o valor booleano encapsulado pela classe.

*método* `compareTo`
- Instâncias da classe `Character` podem ser comparadas numericamente com outras usando o método `compareTo`. Este método recebe uma outra instância da classe como argumento e retorna $-1$ se o caracter encapsulado for menor do que o passado como argumento, $+1$ se for maior e 0 se forem iguais.

*método* `equals`
- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` somente se a instância passada como argumento for da classe `Character` e os valores encapsulados forem os mesmos.

*método* `isDigit`
- O método estático `isDigit` recebe como argumento um valor do tipo nativo `char` e retorna `true` se este caracter for um dígito Unicode. Para a maioria dos programadores, basta saber

que se o caracter passado for um dos `'0'`, `'1'`, `'2'` ... `'9'` o método retornará `true`, mas este método também pode identificar dígitos usados em alfabetos não-romanos.

- O método estático `isLetter` recebe como argumento um `char` e retorna `true` se este caracter for uma letra Unicode. Para a maioria dos programadores basta saber que este método retornará `true` para todas as letras usadas no alfabeto (romano e outros), independente de estarem em maiúsculas ou não. *método* `isLetter`
- O método estático `isLowerCase` recebe como argumento um valor do tipo `char` e retorna `true` se o argumento for um caracter minúsculo (para o alfabeto romando, um dos caracteres entre `'a'` e `'z'`). *método* `isLowerCase`
- O método estático `isUpperCase` recebe como argumento um valor do tipo `char` e retorna `true` se o argumento for um caracter maiúsculo (para o alfabeto romando, um dos caracteres entre `'A'` e `'Z'`). *método* `isUpperCase`
- O método estático `isWhiteSpace` retorna `true` se o argumento passado (do tipo `char`) for um espaço. Caracteres como tabulação, quebra de linha ou página e outros separadores mais obscuros também são considerados espaços por este método. *método* `isWhiteSpace`
- O método estático `toLowerCase` recebe como argumento um valor do tipo `char` e retorna este caracter convertido para minúsculas. Se a conversão for desnecessária ou se não houver uma versão minúscula deste caracter, o próprio argumento é retornado. *método* `toLowerCase`
- O método estático `toUpperCase` recebe como argumento um valor do tipo `char` e retorna este caracter convertido para maiúsculas. Se a conversão for desnecessária ou se não houver uma versão maiúscula deste caracter, o próprio argumento é retornado. *método* `toUpperCase`
- O método estático `toString` retorna uma instância da classe `String` contendo o caracter encapsulado pela instância que executar o método. *método* `toString`

Alguns exemplos de uso de instâncias da classe `Character` são mostrados no programa na listagem A.19.

Listagem A.19: A classe `DemoClasseCharacter`, que demonstra usos da classe `Character`.

```
1  /**
2   * A classe DemoClasseCharacter, que demonstra vários usos corretos da classe
3   * Character.
4   */
5  class DemoClasseCharacter // declaração da classe
6    {
7   /**
8    * O método main permite a execução desta classe. Este método contém declarações
9    * de várias instâncias da classe Character, que encapsula um valor do tipo nativo
10   * char. O método também demonstra algumas operações com estas instâncias.
11   * @param argumentos os argumentos que podem ser passados para o método via linha
12   *        de comando, mas que neste caso serão ignorados.
13   */
14   public static void main(String[] argumentos)
15     {
16     Character c1,c2,c3,c4,c5;      // declaração das instâncias
17     c1 = new Character('a');       // c1 representará o Character 'a'.
18     c2 = new Character((char)65); // c2 representará o Character 'A' - o valor passado
19                                    // como argumento para o construtor da classe
20                                    // Character deve ser explicitamente convertido
21                                    // para um valor do tipo char.
22     c3 = new Character('!');       // c3 representará o Character '!'.
23     c4 = new Character('9');       // c4 representará o Character '9'.
24     c5 = new Character(' ');       // c5 representará o Character ' ' (espaço).
25     char ch1 = c2.charValue();     // recupera o valor encapsulado por c2 como um valor
26                                    // do tipo char
27     // Compara dois Characters numericamente. Note que não podemos comparar estes
28     // valores diretamente com ==, !=  <, >=, etc. O resultado da comparação é baseado
```

```
29      // na ordem dos caracteres na tabela Unicode.
30      if (c1.compareTo(c2) > 0) System.out.println(c1+" é maior que "+c2);
31      if (c3.compareTo(c3) == 0) System.out.println(c3+" é diferente de "+c4);
32      // A classe Character tem vários métodos estáticos para processamento de valores
33      // do tipo nativo char.
34      char a1 = Character.toUpperCase('a'); // conversão para maiúsculo, a1 valerá 'A'
35      char a2 = Character.toLowerCase('A'); // conversão para minúsculo, a2 valerá 'a'
36      // Podemos verificar a categoria dos caracteres usando métodos estáticos.
37      // Os caracteres são dígitos ?
38      System.out.println(Character.isDigit('1')); // true
39      System.out.println(Character.isDigit('Y')); // false
40      System.out.println(Character.isDigit('+')); // false
41      System.out.println(Character.isDigit('/')); // false
42      // Os caracteres são letras ?
43      System.out.println(Character.isLetter('1')); // false
44      System.out.println(Character.isLetter('Y')); // true
45      System.out.println(Character.isLetter('+')); // false
46      System.out.println(Character.isLetter('/')); // false
47      // Os caracteres são letras ou dígitos ?
48      System.out.println(Character.isLetterOrDigit('1')); // true
49      System.out.println(Character.isLetterOrDigit('Y')); // true
50      System.out.println(Character.isLetterOrDigit('+')); // false
51      System.out.println(Character.isLetterOrDigit('/')); // false
52      // Os caracteres são minúsculos ?
53      System.out.println(Character.isLowerCase('m')); // true
54      System.out.println(Character.isLowerCase('M')); // false
55      System.out.println(Character.isLowerCase(' ')); // false
56      System.out.println(Character.isLowerCase('!')); // false
57      // Os caracteres são maiúsculos ?
58      System.out.println(Character.isUpperCase('m')); // false
59      System.out.println(Character.isUpperCase('M')); // true
60      System.out.println(Character.isUpperCase(' ')); // false
61      System.out.println(Character.isUpperCase('!')); // false
62      // Os caracteres são espaços genéricos ?
63      System.out.println(Character.isWhitespace(' '));  // true
64      System.out.println(Character.isWhitespace('M'));  // false
65      System.out.println(Character.isWhitespace('y'));  // false
66      System.out.println(Character.isWhitespace('\t')); // true
67      System.out.println("Faixa de valores do byte:"+// impressão dos valores-limite
68                         (int)Character.MIN_VALUE+   // Note que os valores foram
69                         " a "+                      // convertidos explicitamente para
70                         (int)Character.MAX_VALUE);  // para valores do tipo int para
71                                                     // evitar que os caracteres
72                                                     // correspondentes a 0 e 65535
73                                                     // fossem impressos - provavelmente
74                                                     // não poderiam ser visualizados
75                                                     // corretamente nos terminais.
76      }
77
78    } // fim da classe DemoClasseCharacter
```

Vários métodos de classificação dos caracteres em categorias foram mostrados na listagem A.19 - estes métodos podem ser úteis para aplicações de análise de texto.

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.20.

Listagem A.20: A classe `DemoClasseCharacterComErros`, que demonstra usos incorretos da classe `Character`.

```
1  /**
2   * A classe DemoClasseCharacterComErros, que demonstra vários usos incorretos e algumas
3   * armadilhas da classe Character.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoClasseCharacterComErros // declaração da classe
7    {
```

```
8  /**
9   * O método main permite a execução desta classe. Este método contém declarações de
10  * várias instâncias da classe Character, que encapsula um valor do tipo nativo char.
11  * O método também demonstra algumas operações com estas instâncias. Muitas das
12  * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
13  * compilado.
14  * @param argumentos os argumentos que podem ser passados para o método via linha
15  *        de comando, mas que neste caso serão ignorados.
16  */
17  public static void main(String[] argumentos)
18    {
19    Character c1,c2,c3,c4,c5;      // declaração das instâncias
20    c1 = new Character("a");       // ERRO: o valor passado como argumento para o
21                                   //       construtor da classe Character é uma
22                                   //       instância da classe String e não um valor
23                                   //       do tipo char (note as aspas duplas).
24    c2 = new Character(91);        // ERRO: o valor passado como argumento para o
25                                   //       construtor da classe Character é um valor
26                                   //       do tipo int, devendo ser convertido
27                                   //       explicitamente para o tipo char.
28    c3 = new Character();          // ERRO: não existe construtor vazio (que não receba
29                                   //       argumentos) para a classe Character.
30    c4 = new Character('');        // ERRO: o valor passado como argumento para o
31                                   //       construtor não é um caracter válido.
32    c5 = new Character((char)-1); // não existe erro, mas o caracter encapsulado será
33                                   // o de código 65535
34    // Compara Characters com instâncias de outras classes, com o valor null e com
35    // valores de tipos nativos.
36    if (c1.equals(null)) System.out.println(c1+" é nulo.");  // nada será impresso
37    if (c2.equals(91)) System.out.println(c1+" vale 91."); // ERRO: é impossível
38                                                           // comparar argumentos que
39                                                           // não sejam instâncias com
40                                                           // o método equals.
41    Short s = new Short(91); // criamos uma instância da classe Short
42    if (c2.equals(s))                          // nada será impresso, pois embora c2 e s
43      System.out.println(b2+" é igual a "+i); // sejam instâncias, são instâncias de
44                                             // classes diferentes, não podendo ser
45                                             // comparadas - o resultado será sempre
46                                             // false.
47    }
48
49  } // fim da classe DemoClasseCharacterComErros
```

Nas linhas 20 a 30 da listagem A.20 temos exemplos de argumentos inválidos passados ao construtor da classe `Character`: uma string (linha 20), um valor inteiro sem *cast* (linha 24), nenhum argumento (linha 28) e um caracter vazio (linha 30). O construtor mostrado na linha 32 não terá erros de compilação nem execução, e a instância `c5` representará o caracter cujo código Unicode é 65516.

### A.3.3 A classe `Byte`

*classe* `Byte`

A classe `Byte` encapsula valores do tipo nativo `byte`. Os construtores da classe permitem a inicialização dos valores encapsulados através da especificação de um valor do tipo `byte` ou de uma string que represente um valor do tipo `byte`. Neste segundo caso, a string deve conter somente e exatamente os caracteres necessários para que um byte seja representado.

Outros métodos e campos interessantes da classe `Byte` são:

- O campo estático `MIN_VALUE` representa o menor valor que pode ser armazenado em um campo do tipo `byte`, e é igual a $-128$. *campo* `MIN_VALUE`
- O campo estático `MAX_VALUE` representa o maior valor que pode ser armazenado em um campo do tipo `byte`, e é igual a 127. *campo* `MAX_VALUE`

*método* `decode`
- O método `decode` decodifica a string que é passada como argumento, retornando uma nova instância da classe `Byte`. Este método tenta reconhecer automaticamente a base usada para conversão: se após um sinal negativo opcional a string começar em "`0x`" ou '`#`' a base será hexadecimal, se começar por um '`0`' será octal, caso contrário será considerada decimal. Caso a string contenha espaços ou seus caracteres não sejam compatíveis com a base reconhecida, a exceção `NumberFormatException` ocorrerá.

*método* `parseByte`
- O método estático `parseByte` tem duas formas: uma recebe como argumento uma string que deve conter um `byte`, representado usando a base decimal, e a outra versão recebe como argumento adicional um valor do tipo `int` que será a base usada para conversão do `byte` representado pela string. O método retorna um valor do tipo `byte` contendo o valor decodificado, ou lança a exceção `NumberFormatException` se a string conter espaços, caracteres não reconhecidos como sendo da base usada ou estiver formatada de uma maneira não reconhecível pelo método.

*método* `valueOf`
- O método estático `valueOf` tem duas formas que se comportam da mesma maneira que as do método `parseByte`, exceto que o método retorna uma instância da classe `Byte` encapsulando o valor que foi passado como um argumento (instância da classe `String`). O método também causará a exceção `NumberFormatException` se a string não for reconhecida como um `byte`.

*método* `byteValue`
- O método `byteValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `byte`. Este é o método comumente usado para "desencapsular" um valor encapsulado em uma instância da classe `Byte`.

*método* `shortValue`
- O método `shortValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `short`.

*método* `intValue`
- O método `intValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `int`.

*método* `longValue`
- O método `longValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `long`.

*método* `floatValue`
- O método `floatValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `float`.

*método* `doubleValue`
- O método `doubleValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `double`.

*método* `compareTo`
- O método `compareTo` compara a instância da classe `Byte` com outra, que deve ser passada como um argumento para o método, retornando $-1$ se o valor encapsulado for menor que o valor passado como argumento, $+1$ se for maior e 0 se ambos forem iguais.

*método* `equals`
- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` se esta instância for da classe `Byte` e o valor encapsulado pela instância passada como argumento for igual ao valor encapsulado.

*método* `toString`
- O método `toString` retorna uma string contendo o valor encapsulado. Este método pode ser chamado implicitamente, da mesma forma que outros métodos `toString` de outras classes.

Alguns exemplos de uso de instâncias da classe `Byte` são mostrados no programa na listagem A.21.

Listagem A.21: A classe `DemoClasseByte`, que demonstra usos da classe `Byte`.

```
1 /**
2  * A classe DemoClasseByte, que demonstra vários usos corretos da classe Byte.
3  */
4 class DemoClasseByte // declaração da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
5   {
6  /**
7   * O método main permite a execução desta classe. Este método contém declarações
8   * de várias instâncias da classe Byte, que encapsula um valor do tipo nativo byte.
9   * O método também demonstra algumas operações com estas instâncias.
10  * @param argumentos os argumentos que podem ser passados para o método via linha
11  *        de comando, mas que neste caso serão ignorados.
12  */
13  public static void main(String[] argumentos)
14    {
15    Byte b1,b2;              // declaração das instâncias
16    b1 = new Byte((byte)12); // b1 representará o Byte 12. Note que a conversão
17                             // explícita é necessária: não podemos construir um Byte
18                             // passando um valor do tipo int como argumento.
19    b2 = new Byte("-45");    // b2 representará o Byte -45 - a conversão de string
20                             // para Byte será feita automaticamente.
21    byte b = b1.byteValue(); // recupera o valor encapsulado por b1 como um valor do
22                             // tipo byte
23    int i = b2.intValue();   // recupera o valor encapsulado por b2 como um valor do
24                             // tipo int
25    // Compara dois Bytes numericamente. Note que não podemos comparar estes valores
26    // diretamente com ==, !=  <, >=, etc.
27    if (b1.compareTo(b2) < 0) System.out.println(b1+" é menor que "+b2);
28    Byte b3 = Byte.valueOf("7f",16);      // b3 representará o Byte 127, convertido da
29                                          // string que representa este valor na base
30                                          // hexadecimal
31    Byte b4 = Byte.valueOf("00101101",2); // b4 representará o Byte 45, convertido da
32                                          // string que representa este valor na base
33                                          // binária
34    byte oito = Byte.parseByte("8");      // decodifica uma string contendo um valor do
35                                          // tipo byte, na base decimal, diretamente,
36                                          // sem criar uma instância da classe Byte.
37    byte vinte = Byte.parseByte("14",16); // decodifica uma string contendo um valor do
38                                          // tipo byte, na base hexadecimal, sem criar
39                                          // uma instância da classe Byte.
40    byte cem = Byte.parseByte("1100100",2); // decodifica uma string contendo um valor
41                                            // do tipo byte, na base binária, sem
42                                            // criar uma instância da classe Byte.
43    System.out.println("Faixa de valores do byte:"+ // impressão dos valores-limite
44                       Byte.MIN_VALUE+" a "+
45                       Byte.MAX_VALUE);
46    }
47
48  } // fim da classe DemoClasseByte
```

Podemos ver nas linhas 16 e 19 que instâncias da classe `Byte` podem ser inicializadas com valores do tipo `byte` e strings que contenham valores compatíveis. Nas linhas 21 e 23 vemos exemplos de conversão da instância da classe para tipos nativos. Nas linhas 28 a 40 vemos exemplos de criação de instâncias das classes (sem ser via construtores) e inicialização de variáveis do tipo `byte` usando métodos apropriados que recebem com argumentos strings e opcionalmente bases numéricas diferentes de 10.

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.22.

Listagem A.22: A classe `DemoClasseByteComErros`, que demonstra usos incorretos da classe `Byte`.

```
1  /**
2   * A classe DemoClasseByteComErros, que demonstra vários usos incorretos e algumas
3   * armadilhas da classe Byte.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoClasseByteComErros // declaração da classe
7    {
8   /**
```

```
9   * O método main permite a execução desta classe. Este método contém declarações de
10  * várias instâncias da classe Byte, que encapsula um valor do tipo nativo byte.
11  * O método também demonstra algumas operações com estas instâncias. Muitas das
12  * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
13  * compilado.
14  * @param argumentos os argumentos que podem ser passados para o método via linha
15  *        de comando, mas que neste caso serão ignorados.
16  */
17  public static void main(String[] argumentos)
18    {
19    Byte b1,b2,b3,b4,b5,b6;    // declaração das instâncias
20    b1 = new Byte(200);        // ERRO: o valor passado como argumento é implicitamente
21                               //       um valor do tipo inteiro, devendo ser convertido
22                               //       explicitamente para byte
23    b2 = new Byte((byte)400); // não existe erro, mas o valor convertido
24                               // explicitamente para byte perderá precisão na
25                               // conversão. b2 encapsulará o valor byte -112.
26    b3 = (byte)123;            // ERRO: valor do tipo nativo byte não pode ser
27                               //       convertido para instância da classe Byte
28    b4 = new Byte("-2048");    // ERRO: a string passada como argumento para o
29                               //       construtor contém um valor que não pode ser
30                               //       representado por um byte
31    b5 = new Byte(" 120");     // ERRO: a string passada como argumento para o
32                               //       construtor contém caracteres que não permitem a
33                               //       conversão
34    b6 = new Byte();           // ERRO: não existe construtor vazio (que não receba
35                               //       argumentos) para a classe Byte.
36    // Compara Bytes com instâncias de outras classes, com o valor null e com valores
37    // de tipos nativos.
38    if (b4.equals(null)) System.out.println(b4+" é nulo.");  // nada será impresso
39    if (b1.equals(200)) System.out.println(b1+" vale 200."); // ERRO: é impossível
40                                                             // comparar argumentos que
41                                                             // não sejam instâncias
42                                                             // com o método equals.
43    Integer i = new Integer(-112); // criamos uma instância da classe Integer
44    if (b2.equals(i))                          // nada será impresso, pois embora b2 e i
45      System.out.println(b2+" é igual a "+i); // sejam instâncias, são instâncias de
46                                             // classes diferentes, não podendo ser
47                                             // comparadas - o resultado será sempre
48                                             // false.
49    if (b2.compareTo(i) == 0)                  // ERRO: não será possível efetuar a
50      System.out.println(b2+" é igual a "+i); //       comparação entre classes
51                                             //       diferentes.
52    Byte b7 = Byte.valueOf("ff",16);           // ERRO: o valor contido na string e na
53                                             //       base especificada não pode ser
54                                             //       representado por um byte.
55    Byte b8 = Byte.valueOf("111000111000",2); // ERRO: o mesmo do exemplo acima.
56    Byte b9 = Byte.valueOf("1a",10);           // ERRO: o valor contido na string não
57                                             //       pode ser convertido para byte
58                                             //       usando a base especificada.
59    }
60
61  } // fim da classe DemoClasseByteComErros
```

Na linha 20 da listagem A.22 temos um erro porque o valor 200 é considerado implicitamente como um valor do tipo `int`, e não existe um construtor correspondente na classe `Byte` - é necessário o *cast* do valor 200 para o tipo `byte`. Nas linhas 28 e 31 teremos erros de conversão (em tempo de execução) pois as strings não contém valores válidos para a conversão para bytes. Na linha 34, teremos outro erro de conversão: não existe o construtor sem argumentos na classe `Byte`. As linhas 44 e 49 mostram erros de comparação, descritos nos comentários próximos destas linhas. Finalmente, as linhas 52 a 56 mostram erros usando os métodos de conversão de bases.

### A.3.4 A classe `Short`

*classe* `Short`

A classe `Short` encapsula valores do tipo nativo `short`. Os construtores da classe permitem a inicialização dos valores encapsulados recebendo como argumento um valor do tipo `short` ou uma string que represente um valor do tipo `short`. Neste segundo caso, a string deve conter somente e exatamente os caracteres necessários para que um valor do tipo `short` seja representado.

Outros métodos e campos interessantes da classe `Short` são:

- O campo estático `MIN_VALUE` representa o menor valor que pode ser armazenado em um campo do tipo `short`, e é igual a $-32768$. *campo* `MIN_VALUE`
- O campo estático `MAX_VALUE` representa o maior valor que pode ser armazenado em um campo do tipo `short`, e é igual a 32767. *campo* `MAX_VALUE`
- O método `decode` decodifica a string que é passada como argumento, retornando uma nova instância da classe `Short`. Este método tenta reconhecer automaticamente a base usada para conversão: se após um sinal negativo opcional a string começar em `"0x"` ou `'#'` a base será hexadecimal, se começar por um `'0'` será octal, caso contrário será considerada decimal. Caso a string contenha espaços ou seus caracteres não sejam compatíveis com a base reconhecida, a exceção `NumberFormatException` ocorrerá. *método* `decode`
- O método estático `parseShort` tem duas formas: uma recebe como argumento uma string que deve conter um `short`, representado usando a base decimal, e a outra versão recebe como argumento adicional um valor do tipo `int` que será a base usada para conversão do `short` representado pela string. O método retorna um valor do tipo `short` contendo o valor decodificado, ou lança a exceção `NumberFormatException` se a string conter espaços, caracteres não reconhecidos como sendo da base usada ou estiver formatada de uma maneira não reconhecível pelo método. *método* `parseShort`
- O método estático `valueOf` tem duas formas que se comportam da mesma maneira que as do método `parseShort`, exceto que o método retorna uma instância da classe `Short` encapsulando o valor que foi passado como um argumento (instância da classe `String`). O método também causará a exceção `NumberFormatException` se a string não for reconhecida como um `short`. *método* `valueOf`
- O método `byteValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `byte`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `short` para um valor do tipo `byte`, ou seja, os oito bits mais significativos serão descartados. *método* `byteValue`
- O método `shortValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `short`. Este é o método comumente usado para "desencapsular" um valor encapsulado em uma instância da classe `Short`. *método* `shortValue`
- O método `intValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `int`. *método* `intValue`
- O método `longValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `long`. *método* `longValue`
- O método `floatValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `float`. *método* `floatValue`
- O método `doubleValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `double`. *método* `doubleValue`
- O método `compareTo` compara a instância da classe `Short` com outra, que deve ser passada como um argumento para o método, retornando $-1$ se o valor encapsulado for menor que o valor passado como argumento, $+1$ se for maior e 0 se ambos forem iguais. *método* `compareTo`



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

*método* `equals`

- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` se esta instância for da classe `Short` e o valor encapsulado pela instância passada como argumento for igual ao valor encapsulado.

*método* `toString`

- O método `toString` retorna uma string contendo o valor encapsulado. Este método pode ser chamado implicitamente, da mesma forma que outros métodos `toString` de outras classes.

Alguns exemplos de uso de instâncias da classe `Short` são mostrados no programa na listagem A.23.

Listagem A.23: A classe `DemoClasseShort`, que demonstra usos da classe `Short`.

```
/**
 * A classe DemoClasseShort, que demonstra vários usos corretos da classe Short.
 */
class DemoClasseShort // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações
  * de várias instâncias da classe Short, que encapsula um valor do tipo nativo short.
  * O método também demonstra algumas operações com estas instâncias.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    Short s1,s2;                  // declaração das instâncias
    s1 = new Short((short)9999); // s1 representará o Short 9999. Note que a conversão
                                  // explícita é necessária: não podemos construir um
                                  // Short passando um valor do tipo int como argumento.
    s2 = new Short("-1024");      // s2 representará o Short -1024 - a conversão de
                                  // string para Short será feita automaticamente.
    short s = s1.shortValue();    // recupera o valor encapsulado por s1 como um valor
                                  // do tipo short
    float f = s2.floatValue();    // recupera o valor encapsulado por s2 como um valor
                                  // do tipo float
    // Compara dois Shorts numericamente. Note que não podemos comparar estes valores
    // diretamente com ==, !=  <, >=, etc.
    if (s1.compareTo(s2) > 0) System.out.println(s1+" é maior que "+s2);
    Short s3 = Short.valueOf("1234",16);  // s3 representará o Short 4660, convertido
                                          // da string que representa este valor na
                                          // base hexadecimal
    Short s4 = Short.valueOf("11111100000000",2); // s4 representará o Short 16128,
                                                  // convertido da string que
                                                  // representa este valor na base
                                                  // binária
    short quinze = Short.parseShort("15"); // decodifica uma string contendo um valor
                                           // do tipo short, na base decimal,
                                           // diretamente, sem criar uma instância da
                                           // classe Short.
    short mil = Short.parseShort("rs",36); // decodifica uma string contendo um valor
                                           // do tipo short, na base 36, sem criar
                                           // uma instância da classe Short.
    short menosMil = Short.parseShort("-1111101000",2); // decodifica uma string
                                                        // contendo um valor do tipo
                                                        // short, na base binária, sem
                                                        // criar uma instância da
                                                        // classe Short.
    System.out.println("Faixa de valores do short:"+ // impressão dos valores-limite
                       Short.MIN_VALUE+" a "+
                       Short.MAX_VALUE);
    }

  } // fim da classe DemoClasseShort
```

As linhas 16 e 19 da listagem A.23 mostram usos dos construtores da classe `Short`, que aceitam respectivamente valores do tipo `short` e strings contendo valores compatíveis. As linhas 21 e 23 mostram como podemos recuperar um valor armazenado em uma instância da classe `Short` como valores dos tipos `short` e `float`, respectivamente.

As linhas 28 a 42 mostram como podemos converter valores contidos em strings para instâncias da classe `Short` ou diretamente variáveis do tipo `short`, usando bases numéricas diferentes. É possível o uso de qualquer base numérica entre 2 e 36 (inclusive), como mostrado nos exemplos. A base 36 usa os números `0` a `9` e os caracteres `A` a `Z` (maiúsculos ou minúsculos).

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.24.

Listagem A.24: A classe `DemoClasseShortComErros`, que demonstra usos incorretos da classe `Short`.

```
1  /**
2   * A classe DemoClasseShortComErros, que demonstra vários usos incorretos e algumas
3   * armadilhas da classe Short.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoClasseShortComErros // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias instâncias da classe Short, que encapsula um valor do tipo nativo short.
11   * O método também demonstra algumas operações com estas instâncias. Muitas das
12   * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
13   * compilado.
14   * @param argumentos os argumentos que podem ser passados para o método via linha
15   *        de comando, mas que neste caso serão ignorados.
16   */
17   public static void main(String[] argumentos)
18     {
19     Short s1,s2,s3,s4,s5,s6,s7;   // declaração das instâncias
20     s1 = new Short(200);          // ERRO: o valor passado como argumento é
21                                   //       implicitamente um valor do tipo inteiro,
22                                   //       devendo ser convertido explicitamente para
23                                   //       short.
24     s2 = new Short((short)99999); // não existe erro, mas o valor convertido
25                                   // explicitamente para short perderá precisão na
26                                   // conversão. s2 encapsulará o valor short -31073
27     s3 = (short)99;               // ERRO: valor do tipo nativo short não pode ser
28                                   //       convertido para instância da classe Short.
29     s4 = new Short("77777");      // ERRO: a string passada como argumento para o
30                                   //       construtor contém um valor que não pode ser
31                                   //       representado por um short.
32     s5 = new Short("1 2 3");      // ERRO: a string passada como argumento para o
33                                   //       construtor contém caracteres que não
34                                   //       permitem a conversão
35     s6 = new Short();             // ERRO: não existe construtor vazio (que não receba
36                                   //       argumentos) para a classe Short.
37     s7 = new Short((short)130);   // não existe erro, mas...
38     byte b = s7.byteValue();      // a variável b valerá -126, pois precisão será
39                                   // perdida quando convertermos um valor do tipo
40                                   // short para byte.
41     // Compara Shorts com instâncias de outras classes, com o valor null e com valores
42     // de tipos nativos.
43     if (s4.equals(null)) System.out.println(s4+" é nulo.");  // nada será impresso
44     if (s3.equals(99)) System.out.println(s3+" vale 1500."); // ERRO: é impossível
45                                                              // comparar argumentos que
46                                                              // não sejam instâncias
47                                                              // com o método equals.
48     Integer i = new Integer(-31073); // criamos uma instância da classe Integer
49     if (s2.equals(i))                        // nada será impresso, pois embora s2 e i
```

```
50        System.out.println(s2+" é igual a "+i); // sejam instâncias, são instâncias de
51                                                // classes diferentes, não podendo ser
52                                                // comparadas - o resultado será sempre
53                                                // false.
54      if (s2.compareTo(i) == 0)                 // ERRO: não será possível efetuar a
55        System.out.println(s2+" é igual a "+i); //       comparação entre classes
56                                                //       diferentes.
57      Short s8 = Short.valueOf("8000",16);      // ERRO: o valor contido na string e na
58                                                //       base especificada não pode ser
59                                                //       representado por um short.
60      Short s9 = Short.valueOf("1111111111111111",2); // ERRO: o valor contido na string
61                                                      //       não pode ser convertido
62                                                      //       para um short: o maior
63                                                      //       valor positivo binário
64                                                      //       que pode ser convertido é
65                                                      //       0111111111111111.
66      Short s10 = Short.valueOf("1099",8);      // ERRO: o valor contido na string não
67                                                //       pode ser convertido para short
68                                                //       usando a base especificada.
69      }
70
71  } // fim da classe DemoClasseShortComErros
```

Na linha 20 da listagem A.24 vemos uma chamada ao construtor da classe `Short` com um valor de tipo não válido (não existe construtor para o tipo `int`). Os valores passados nas strings das linhas 29 e 32 não podem ser transformados em valores do tipo `short`, causando erros de execução (a string mostrada na linha 32 contém espaços entre os caracteres numéricos). A conversão mostrada na linha 66 causará erro de execução porque a base somente reconhecerá os caracteres de `'0'` a `'7'`, e outros caracteres estão presentes na string. Outros erros de conversão e comparação são descritos nos comentários da listagem.

### A.3.5 A classe `Integer`

*classe* `Integer`

A classe `Integer` encapsula valores do tipo nativo `int`. Os construtores da classe permitem a inicialização dos valores encapsulados recebendo como argumentos um valor do tipo `int` ou uma string que contenha um valor do tipo `int`. Neste segundo caso, a string deve conter somente e exatamente os caracteres necessários para que um valor do tipo `int` seja representado.

Além dos construtores, outros métodos e campos úteis da classe `Integer` são:

*campo* `MIN_VALUE`

- O campo estático `MIN_VALUE` representa o menor valor que pode ser armazenado em um campo do tipo `int`, e é igual a $-2147483648$.

*campo* `MAX_VALUE`

- O campo estático `MAX_VALUE` representa o maior valor que pode ser armazenado em um campo do tipo `int`, e é igual a 2147483647.

*método* `decode`

- O método `decode` decodifica a string que é passada como argumento, retornando uma nova instância da classe `Integer`. Este método tenta reconhecer automaticamente a base usada para conversão: se após um sinal negativo opcional a string começar em `"0x"` ou `'#'` a base será hexadecimal, se começar por um `'0'` será octal, caso contrário será considerada decimal. Caso a string contenha espaços ou seus caracteres não sejam compatíveis com a base reconhecida, a exceção `NumberFormatException` ocorrerá.

*método* `parseInt`

- O método estático `parseInt` tem duas formas: uma recebe como argumento uma string que deve conter um `int`, representado usando a base decimal, e a outra versão recebe como argumento adicional um valor do tipo `int` que será a base usada para conversão do `int` representado pela string. O método retorna um valor do tipo `int` contendo o valor decodificado, ou lança a exceção `NumberFormatException` se a string conter espaços, caracteres não reconhecidos como sendo da base usada ou estiver formatada de uma maneira não reconhecível pelo método.

- O método estático `valueOf` tem duas formas que se comportam da mesma maneira que as do método `parseInt`, exceto que o método retorna uma instância da classe `Integer` encapsulando o valor que foi passado como um argumento (instância da classe `String`). O método também causará a exceção `NumberFormatException` se a string não for reconhecida como um `int`. *método* `valueOf`
- O método `byteValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `byte`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `int` para um valor do tipo `byte`, ou seja, os 24 bits mais significativos serão descartados. *método* `byteValue`
- O método `shortValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `short`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `int` para um valor do tipo `short`, ou seja, os 16 bits mais significativos serão descartados. *método* `shortValue`
- O método `intValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `int`. Este é o método comumente usado para "desencapsular" um valor encapsulado em uma instância da classe `Integer`. *método* `intValue`
- O método `longValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `long`. *método* `longValue`
- O método `floatValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `float`. *método* `floatValue`
- O método `doubleValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `double`. *método* `doubleValue`
- O método `compareTo` compara a instância da classe `Integer` com outra, que deve ser passada como um argumento para o método, retornando $-1$ se o valor encapsulado for menor que o valor passado como argumento, $+1$ se for maior e 0 se ambos forem iguais. *método* `compareTo`
- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` se esta instância for da classe `Integer` e o valor encapsulado pela instância passada como argumento for igual ao valor encapsulado. *método* `equals`
- O método `toString` retorna uma string contendo o valor encapsulado. Este método pode ser chamado implicitamente, da mesma forma que outros métodos `toString` de outras classes. Este método tem duas outras formas (que não podem ser chamadas implicitamente) que são declaradas como `static`: uma recebe como argumento um valor inteiro e retorna uma string representando aquele valor e outra recebe um argumento adicional (do tipo `int`) que é a base usada na conversão de `int` para string. *método* `toString`
- O método `toBinaryString` converte o valor do tipo `int` passado como argumento para uma string, usando a base binária para conversão. Este método equivale ao método `toString` com dois argumentos, passando 2 como segundo argumento. *método* `toBinaryString`
- O método `toOctalString` converte o valor do tipo `int` passado como argumento para uma string, usando a base octal para conversão. Este método equivale ao método `toString` com dois argumentos, passando 8 como segundo argumento. *método* `toOctalString`
- O método `toHexString` converte o valor do tipo `int` passado como argumento para uma string, usando a base hexadecimal para conversão. Este método equivale ao método `toString` com dois argumentos, passando 16 como segundo argumento. *método* `toHexString`

Alguns exemplos de uso de instâncias da classe `Integer` são mostrados no programa na listagem A.25.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem A.25: A classe `DemoClasseInteger`, que demonstra usos da classe `Integer`.

```
1  /**
2   * A classe DemoClasseInteger, que demonstra vários usos corretos da classe Integer.
3   */
4  class DemoClasseInteger // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias instâncias da classe Integer, que encapsula um valor do tipo nativo int.
9    * O método também demonstra algumas operações com estas instâncias.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     Integer i1,i2;              // declaração das instâncias
16     i1 = new Integer(-100000); // i1 representará o int -100000. Neste caso a conversão
17                                // explícita não é necessária: valores numéricos sem
18                                // ponto flutuante são considerados como ints.
19     i2 = new Integer("-8");     // i2 representará o int -8 - a conversão de string para
20                                // int será feita automaticamente.
21     int i = i1.intValue();      // recupera o valor encapsulado por i1 como um valor
22                                // do tipo int
23     double d = i2.doubleValue(); // recupera o valor encapsulado por i2 como um valor
24                                  // do tipo double
25     // Compara dois Integers numericamente. Note que não podemos comparar estes valores
26     // diretamente com ==, !=  <, >=, etc.
27     if (i1.compareTo(i2) == 0) System.out.println(i1+" e "+i2+" são iguais.");
28     Integer i3 = Integer.valueOf("7fffffff",16); // i3 representará o Integer
29                                                  // 2147483647, convertido da string
30                                                  // que representa este valor na base
31                                                  // hexadecimal
32     Integer i4 = Integer.valueOf("-1000",2); // i4 representará o Integer -8,
33                                             // convertido da string que representa
34                                             // este valor na base binária
35     int sessenta = Integer.parseInt("60"); // decodifica uma string contendo um valor
36                                           // do tipo int, na base decimal,
37                                           // diretamente, sem criar uma instância da
38                                           // classe Integer.
39     int valor = Integer.parseInt("xxx",36); // decodifica uma string contendo um valor
40                                             // do tipo int, na base 36 (43989), sem
41                                             // criar uma instância da classe Integer.
42     String convertido; // declaração de uma string que vai receber o resultado da
43                        // conversão de um inteiro para string usando várias bases
44     convertido = Integer.toBinaryString(valor); // converte para binário
45     convertido = Integer.toOctalString(valor);  // converte para octal (base 8)
46     convertido = Integer.toHexString(valor);    // converte para hexadecimal (base 16)
47     convertido = Integer.toString(valor,36);    // converte para base 36
48     System.out.println("Faixa de valores do int:"+ // impressão dos valores-limite
49                        Integer.MIN_VALUE+" a "+
50                        Integer.MAX_VALUE);
51     }
52
53   } // fim da classe DemoClasseInteger
```

Nas linhas 16 e 19 da listagem A.25 vemos a criação de duas instâncias da classe `Integer`, usando um valor inteiro e uma string contendo os caracteres que representam um valor inteiro. Nas linhas 21 e 23 vemos a conversão dos valores contidos em instâncias da classe `Integer` para valores dos tipos `int` e `double`, respectivamente. Nas linhas 28 a 39 temos a conversão de strings em instâncias da classe e em tipos nativos, usando duas bases diferentes.

Nas linhas 44 a 47 temos a conversão dos valores passados para strings usando as bases 2, 8, 16 e 36, usando os métodos estáticos `toBinaryString`, `toOctalString`, `toHexString` e `toString`, respectivamente. As linhas 48 a 50 imprimem os valores mínimo e máximo represen-



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

táveis pelo tipo `int`.

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.26.

Listagem A.26: A classe `DemoClasseIntegerComErros`, que demonstra usos incorretos da classe `Integer`.

```
/**
 * A classe DemoClasseIntegerComErros, que demonstra vários usos incorretos e algumas
 * armadilhas da classe Integer.
 * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
 */
class DemoClasseIntegerComErros // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * várias instâncias da classe Integer, que encapsula um valor do tipo nativo int.
  * O método também demonstra algumas operações com estas instâncias. Muitas das
  * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
  * compilado.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *       de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    Integer i1,i2,i3,i4,i5,i6,i7;   // declaração das instâncias
    i1 = new Short((short)200);     // ERRO: não é possível inicializar uma instância
                                    //       da classe Integer com uma instância da
                                    //       classe Short (ou de qualquer outra).
    i2 = new Integer(3000000000L); // ERRO: não é possível converter do tipo long para
                                    //       o tipo int, esperado pelo construtor da
                                    //       classe Integer.
    i3 = 1000000000;                // ERRO: valor do tipo nativo int não pode ser con-
                                    //       vertido para instância da classe Integer.
    i4 = new Integer("3000000000"); // ERRO: a string passada como argumento para o
                                    //       construtor contém um valor que não pode
                                    //       ser representado por um int.
    i5 = new Integer("22-17");      // ERRO: a string passada como argumento para o
                                    //       construtor contém caracteres que não
                                    //       permitem a conversão.
    i6 = new Integer();             // ERRO: não existe construtor vazio (que não
                                    //       receba argumentos) para a classe Integer
    i7 = new Integer(999999);       // não existe erro, mas...
    short s = i7.shortValue();      // a variável s valerá 16959, pois precisão será
                                    // perdida quando convertermos um valor do tipo
                                    // int para short;
    byte b = i7.byteValue();        // e a variável b valerá 63, por causa da perda
                                    // adicional de precisão.
    // Compara Integers com instâncias de outras classes, com o valor null e com
    // valores de tipos nativos.
    if (i7.equals(null)) System.out.println(i7+" é nulo."); // nada será impresso
    if (i6.equals(0)) System.out.println(i6+" vale 0.");    // ERRO: é impossível
                                                            // comparar argumentos que
                                                            // não sejam instâncias
                                                            // com o método equals.
    Double d = new Double(999999); // criamos uma instância da classe Double
    if (i7.equals(d))                       // nada será impresso, pois embora s2 e i
      System.out.println(i7+" é igual a "+d); // sejam instâncias, são instâncias de
                                              // classes diferentes, não podendo ser
                                              // comparadas - o resultado será sempre
                                              // false.
    if (i7.compareTo(d) == 0)               // ERRO: não será possível efetuar a
      System.out.println(i7+" é igual a "+d); //       comparação entre classes
                                              //       diferentes.
    Integer i8 = Integer.valueOf("fffffffffff",16); // ERRO: o valor contido na string
                                                    //       e na base especificada
                                                    //       não pode ser representado
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
61                                                  //       por um int.
62      Integer i9 = Integer.valueOf("ffffffff",16); // ERRO: o valor contido na string não
63                                                  //       pode ser convertido para um
64                                                  //       int: o maior valor positivo
65                                                  //       hexadecimal que pode ser
66                                                  //       convertido é 7fffffff
67      Integer i10 = Integer.valueOf("abcd",10);   // ERRO: o valor contido na string
68                                                  //       não pode ser convertido para
69                                                  //       int usando a base
70                                                  //       especificada.
71      }
72
73    } // fim da classe DemoClasseIntegerComErros
```

Nas linhas 20, 23, 28, 31 e 34 da listagem A.26 vemos chamadas incorretas a construtores da classe, sendo que os erros das linhas 28 e 31 ocorrerão em tempo de execução, pois as strings usadas como argumentos não poderão ser convertidas para valores do tipo `int` para encapsulamento nas instâncias da classe. As linhas 37 e 40 não mostrarão erros mas darão como resultados valores com perda de informação. As linhas 58, 62 e 67 mostram erros de conversão de bases.

### A.3.6 A classe `Long`

*classe* `Long`

A classe `Long` encapsula valores do tipo nativo `long`. Os construtores da classe permitem a inicialização dos valores encapsulados recebendo como argumentos um valor do tipo `long` ou uma string que represente um valor do tipo `long`. Neste segundo caso, a string deve conter somente e exatamente os caracteres necessários para que um valor do tipo `long` seja representado.

Embora não existam construtores para a classe `Long` que aceitem valores como `int` e `short` como argumentos, como estes valores podem ser convertidos implicitamente para `long`, o compilador não indicará erros.

Outros métodos e campos interessantes da classe `Long` são:

*campo* `MIN_VALUE`
- O campo estático `MIN_VALUE` representa o menor valor que pode ser armazenado em um campo do tipo `long`, e é igual a $-9223372036854775808$.

*campo* `MAX_VALUE`
- O campo estático `MAX_VALUE` representa o maior valor que pode ser armazenado em um campo do tipo `long`, e é igual a $9223372036854775807$.

*método* `decode`
- O método `decode` decodifica a string que é passada como argumento, retornando uma nova instância da classe `Long`. Este método tenta reconhecer automaticamente a base usada para conversão: se após um sinal negativo opcional a string começar em "`0x`" ou '`#`' a base será hexadecimal, se começar por um '`0`' será octal, caso contrário será considerada decimal. Caso a string contenha espaços ou seus caracteres não sejam compatíveis com a base reconhecida, a exceção `NumberFormatException` ocorrerá.

*método* `parseLong`
- O método estático `parseLong` tem duas formas: uma recebe como argumento uma string que deve conter um `long`, representado usando a base decimal, e a outra versão recebe como argumento adicional um valor do tipo `int` que será a base usada para conversão do `long` representado pela string. O método retorna um valor do tipo `long` contendo o valor decodificado, ou lança a exceção `NumberFormatException` se a string conter espaços, caracteres não reconhecidos como sendo da base usada ou estiver formatada de uma maneira não reconhecível pelo método.

*método* `valueOf`
- O método estático `valueOf` tem duas formas que se comportam da mesma maneira que as do método `parseLong`, exceto que o método retorna uma instância da classe `Long` encapsulando o valor que foi passado como um argumento (instância da classe `String`). O



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

método também causará a exceção `NumberFormatException` se a string não for reconhecida como um `long`.

- O método `byteValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `byte`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `long` para um valor do tipo `byte`, ou seja, os 56 bits mais significativos serão descartados. *método* `byteValue`
- O método `shortValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `short`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `long` para um valor do tipo `short`, ou seja, os 48 bits mais significativos serão descartados. *método* `shortValue`
- O método `intValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `int`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `long` para um valor do tipo `int`, ou seja, os 32 bits mais significativos serão descartados. *método* `intValue`
- O método `longValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `long`. Este é o método comumente usado para "desencapsular" um valor encapsulado em uma instância da classe `Long`. *método* `longValue`
- O método `floatValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `float`. *método* `floatValue`
- O método `doubleValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `double`. *método* `doubleValue`
- O método `compareTo` compara a instância da classe `Long` com outra, que deve ser passada como um argumento para o método, retornando $-1$ se o valor encapsulado for menor que o valor passado como argumento, $+1$ se for maior e 0 se ambos forem iguais. *método* `compareTo`
- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` se esta instância for da classe `Long` e o valor encapsulado pela instância passada como argumento for igual ao valor encapsulado. *método* `equals`
- O método `toString` retorna uma string contendo o valor encapsulado. Este método pode ser chamado implicitamente, da mesma forma que outros métodos `toString` de outras classes. Este método tem duas outras formas (que não podem ser chamadas implicitamente) que são declaradas como `static`: uma recebe como argumento um valor do tipo `long` e retorna uma string representando aquele valor e outra recebe um argumento adicional (do tipo `int`) que é a base usada na conversão de `long` para string. *método* `toString`
- O método `toBinaryString` converte o valor do tipo `long` passado como argumento para uma string, usando a base binária para conversão. Este método equivale ao método `toString` com dois argumentos, passando 2 como segundo argumento. *método* `toBinaryString`
- O método `toOctalString` converte o valor do tipo `long` passado como argumento para uma string, usando a base octal para conversão. Este método equivale ao método `toString` com dois argumentos, passando 8 como segundo argumento. *método* `toOctalString`
- O método `toHexString` converte o valor do tipo `long` passado como argumento para uma string, usando a base hexadecimal para conversão. Este método equivale ao método `toString` com dois argumentos, passando 16 como segundo argumento. *método* `toHexString`

Alguns exemplos de uso de instâncias da classe `Long` são mostrados no programa na listagem A.27.



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Listagem A.27: A classe `DemoClasseLong`, que demonstra usos da classe `Long`.

```
1  /**
2   * A classe DemoClasseLong, que demonstra vários usos corretos da classe Long.
3   */
4  class DemoClasseLong // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias instâncias da classe Long, que encapsula um valor do tipo nativo long.
9    * O método também demonstra algumas operações com estas instâncias.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     Long l1,l2,l3,l4;             // declaração das instâncias
16     l1 = new Long(10000000000L); // l1 representará o long 10000000000l. Note que é
17                                  // necessário representar o valor constante que será
18                                  // passado para o construtor da classe Long com o
19                                  // sufixo "L" para que seja reconhecido como um valor
20                                  // do tipo long.
21     l2 = new Long("0");          // l2 representará o long 0 (zero) - a conversão de
22                                  // string para long será feita automaticamente.
23     long l = l1.longValue();     // recupera o valor encapsulado por l1 como um valor
24                                  // do tipo long
25     float f = l2.floatValue();   // recupera o valor encapsulado por l2 como um valor
26                                  // do tipo float
27     // Compara dois Longs numericamente. Note que não podemos comparar estes valores
28     // diretamente com ==, !=  <, >=, etc.
29     if (l1.compareTo(l2) != 0) System.out.println(l1+" e "+l2+" são diferentes.");
30     l3 = Long.valueOf("7fffffffffffffff",16); // l3 representará o Long
31                                               // 9223372036854775807, convertido da
32                                               // string que representa este valor na
33                                               // base hexadecimal
34     l4 = Long.valueOf("10000000000000000",2); // l4 representará o Long 262144,
35                                                // convertido da string que representa
36                                                // este valor na base binária
37     long milhão = Long.parseLong("1000000"); // decodifica uma string contendo um valor
38                                              // do tipo long, na base decimal,
39                                              // diretamente, sem criar uma instância
40                                              // da classe Long.
41     String convertido; // declaração de uma string que vai receber o resultado da
42                        // conversão de um long para string usando várias bases
43     convertido = Long.toBinaryString(milhão); // converte para binário
44     convertido = Long.toOctalString(milhão);  // converte para octal (base 8)
45     convertido = Long.toHexString(milhão);    // converte para hexadecimal (base 16)
46     convertido = Long.toString(milhão,21);    // converte para base 21
47     System.out.println("Faixa de valores do long:"+ // impressão dos valores-limite
48                        Long.MIN_VALUE+" a "+
49                        Long.MAX_VALUE);
50     }
51
52   } // fim da classe DemoClasseLong
```

Nas linhas 16 e 21 da listagem A.27 temos a criação de duas instâncias da classe `Long` usando um valor do tipo `long` e uma string como argumentos para os construtores. Nas linhas 23 e 25 vemos a conversão dos valores contidos nas instâncias para os tipos `long` e `float`, respectivamente. Nas linhas 30, 34 e 37 vemos a conversão de strings para instâncias da classe `Long` e diretamente para um valor do tipo nativo `long` usando diversas bases. Nas linhas 43 a 46 vemos a conversão de valores do tipo `long` para strings usando diversas bases. Finalmente, nas linhas 47 a 49 vemos a impressão dos valores mínimo e máximo para o tipo `long`.

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.28.

Listagem A.28: A classe `DemoClasseLongComErros`, que demonstra usos incorretos da classe `Long`.

```
/**
 * A classe DemoClasseLongComErros, que demonstra vários usos incorretos e algumas
 * armadilhas da classe Long.
 * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
 */
class DemoClasseLongComErros // declaração da classe
  {
 /**
  * O método main permite a execução desta classe. Este método contém declarações de
  * várias instâncias da classe Long, que encapsula um valor do tipo nativo long.
  * O método também demonstra algumas operações com estas instâncias. Muitas das
  * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
  * compilado.
  * @param argumentos os argumentos que podem ser passados para o método via linha
  *        de comando, mas que neste caso serão ignorados.
  */
  public static void main(String[] argumentos)
    {
    Long l1,l2,l3,l4,l5,l6,l7;    // declaração das instâncias
    l1 = new Integer(-1024);      // ERRO: não é possível inicializar uma instância
                                  //       da classe Long com uma instância da
                                  //       classe Integer (ou de qualquer outra).
    l2 = new Long(1e45);          // ERRO: não é possível converter do tipo double para
                                  //       o tipo long, esperado pelo construtor da
                                  //       classe Long.
    l3 = 190l;                    // ERRO: valor do tipo nativo long não pode ser con-
                                  //       vertido para instância da classe Long.
    l4 = new Long("1000000000000000000"); // ERRO: a string passada como argumento
                                          //       para o construtor contém um valor
                                          //       que não pode ser representado por
                                          //       um long.
    l5 = new Long("999*999*99"); // ERRO: a string passada como argumento para o
                                 //       construtor contém caracteres que não
                                 //       permitem a conversão.
    l6 = new Long();             // ERRO: não existe construtor vazio (que não
                                 //       receba argumentos) para a classe Long
    l7 = new Long(9999999999L);  // não existe erro, mas...
    int i = l7.intValue();       // a variável i valerá 1410065407 pois precisão será
                                 // perdida quando convertermos um valor do tipo
                                 // long para int;
    short s = l7.shortValue();   // similarmente a variável s valerá -7169;
    byte b = l7.byteValue();     // e a variável b valerá -1.
    // Compara Longs com instâncias de outras classes, com o valor null e com
    // valores de tipos nativos.
    if (l7.equals(null)) System.out.println(l7+" é nulo."); // nada será impresso
    if (l6.equals(1L)) System.out.println(l6+" vale 1L.");  // ERRO: é impossível
                                                            // comparar argumentos que
                                                            // não sejam instâncias
                                                            // com o método equals.
    Float f = new Float(9999999999.); // criamos uma instância da classe Float
    if (l7.equals(f))                 // nada será impresso, pois embora l7 e f
      System.out.println(l7+" é igual a "+f); // sejam instâncias, são instâncias de
                                              // classes diferentes, não podendo ser
                                              // comparadas - o resultado será sempre
                                              // false.
    if (l7.compareTo(f) == 0)         // ERRO: não será possível efetuar a
      System.out.println(l7+" é igual a "+f); //       comparação entre classes
                                              //       diferentes.
    Long l8 = Long.valueOf("ffffffffffffffffff",16); // ERRO: o valor contido na
                                                     //       string e na base espe-
                                                     //       cificada não pode ser
                                                     //       representado por um
                                                     //       long.
    Long l9 = Long.valueOf("fffffffffffffff",16); // ERRO: o valor contido na string
                                                  //       não pode ser convertido
                                                  //       para um long: o maior valor
```

```
67                                                  //       positivo hexadecimal que
68                                                  //       pode ser convertido é
69                                                  //       7fffffffffffffff
70      Long l10 = Long.valueOf("cafebabe",10);     // ERRO: o valor contido na string
71                                                  //       não pode ser convertido para
72                                                  //       long usando a base
73                                                  //       especificada.
74      }
75
76  } // fim da classe DemoClasseLongComErros
```

Nas linhas 20, 23, 28, 32 e 35 da listagem A.28 vemos chamadas incorretas a construtores ou chamadas a construtores inexistentes para a classe. Nas linhas 45 a 58 vemos erros de comparação de instâncias com outras instâncias e valores, e nas linhas 59, 64 e 70 vemos erros de conversão de strings usando diversas bases.

### A.3.7 A classe `Float`

*classe* `Float`

A classe `Float` encapsula valores do tipo nativo `float`. Os construtores da classe permitem a inicialização dos valores encapsulados através da especificação de um valor do tipo `float` ou `double` ou de uma string que contenha a representação de um valor do tipo `float`.

Se um valor do tipo `double` for passado como argumento para o construtor, e o valor for maior do que o máximo representável por um `float`, a instância da classe representará infinito (positivo ou negativo, dependendo do sinal do argumento para o construtor). Se a string passada como argumento para o construtor não representar um valor do tipo `float`, a exceção `NumberFormatException` ocorrerá.

Outros métodos e campos interessantes da classe `Float` são:

*campo* `MIN_VALUE`
- O campo estático `MIN_VALUE` representa o menor valor que pode ser armazenado em um campo do tipo `float`, e é igual a $1.4E-45$.

*campo* `MAX_VALUE`
- O campo estático `MAX_VALUE` representa o maior valor que pode ser armazenado em um campo do tipo `float`, e é igual a $3.4028235E38$.

*campo* `NaN`
- O campo estático `NaN` representa o valor especial *Not a Number* do tipo `float`. Operações como divisão de zero por zero ou multiplicação de zero por infinito onde os argumentos são do tipo `float` terão como resultado este valor.

*campo* `POSITIVE_INFINITY`
- O campo estático `POSITIVE_INFINITY` representa o valor especial infinito positivo do tipo `float`. Divisões de valores positivos por zero onde os operadores são do tipo `float` terão como resultado este valor.

*campo* `NEGATIVE_INFINITY`
- O campo estático `NEGATIVE_INFINITY` representa o valor especial infinito negativo do tipo `float`. Divisões de valores negativos por zero onde os operadores são do tipo `float` terão como resultado este valor.

*método* `parseFloat`
- O método estático `parseFloat` recebe como argumento uma string que deve representar adequadamente um valor do tipo `float`, e retorna um valor do tipo `float` contendo o valor decodificado, ou lança a exceção `NumberFormatException` se a string conter espaços, caracteres não reconhecidos ou estiver formatada de uma maneira não reconhecível pelo método. Este método reconhece strings no formato *xey* como válidas, onde *x* é um valor com ponto flutuante opcional, precedido de um sinal de menos também opcional, e *y* é um valor inteiro precedido opcionalmente de um sinal de menos.

*método* `valueOf`
- O método estático `valueOf` se comporta da mesma maneira que o método `parseFloat`, exceto que o método retorna uma instância da classe `Float` encapsulando o valor que foi



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

passado como um argumento (instância da classe `String`). O método também causará a exceção `NumberFormatException` se a string não for reconhecida como um `float`.

- O método `byteValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `byte`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `float` para um valor do tipo `byte`. *método* `byteValue`
- O método `shortValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `short`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `float` para um valor do tipo `short`. *método* `shortValue`
- O método `intValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `int`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `float` para um valor do tipo `int`. *método* `intValue`
- O método `longValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `long`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `float` para um valor do tipo `long`. *método* `longValue`
- O método `floatValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `float`. Este é o método comumente usado para "desencapsular" um valor encapsulado em uma instância da classe `Float`. *método* `floatValue`
- O método `doubleValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `double`. *método* `doubleValue`
- O método `compareTo` compara a instância da classe `Float` com outra, que deve ser passada como um argumento para o método, retornando $-1$ se o valor encapsulado for menor que o valor passado como argumento, $+1$ se for maior e 0 se ambos forem iguais. Este método considera o valor `NaN` como sendo maior que todos os outros, até mesmo maior que infinito, e igual somente a si mesmo. *método* `compareTo`
- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` se esta instância for da classe `Float` e o valor encapsulado pela instância passada como argumento for igual ao valor encapsulado. Uma única exceção: este método considera $+0$ e $-0$ (resultados da divisão por infinito, onde os sinais dos operadores determinam os sinais do zero) como sendo diferentes. *método* `equals`
- O método `isInfinite` tem duas formas: se chamado a partir de uma instância da classe `Float` ele retornará `true` se o valor encapsulado pela instância for infinito (positivo ou negativo). A outra forma é estática, recebe como argumento um valor do tipo `float` e retorna `true` se o valor passado como argumento for infinito. *método* `isInfinite`
- O método `isNaN` tem duas formas: se chamado a partir de uma instância da classe `Float` ele retornará `true` se o valor encapsulado pela instância for `NaN`. A outra forma é estática, recebe como argumento um valor do tipo `float` e retorna `true` se o valor passado como argumento for `NaN`. *método* `isNaN`
- O método `toString` retorna uma string contendo o valor encapsulado. Este método pode ser chamado implicitamente, da mesma forma que outros métodos `toString` de outras classes. Este método tem uma outra formas (que não pode ser chamada implicitamente) que é declaradas como `static`, uma recebe como argumento um valor do tipo `float` e retorna uma string representando aquele valor. *método* `toString`



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Alguns exemplos de uso de instâncias da classe `Float` são mostrados no programa na listagem A.29.

Listagem A.29: A classe `DemoClasseFloat`, que demonstra usos da classe `Float`.

```
1  /**
2   * A classe DemoClasseFloat, que demonstra vários usos corretos da classe Float.
3   */
4  class DemoClasseFloat // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações
8    * de várias instâncias da classe Float, que encapsula um valor do tipo nativo float.
9    * O método também demonstra algumas operações com estas instâncias.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     Float f1,f2,f3,f4,f5;      // declaração das instâncias
16     f1 = new Float(1e30);      // f1 representará o float 1*10^30 - a classe Float
17                                // tem construtores sobrecarregados que aceitam como
18                                // argumentos valores do tipo float ou double.
19     f2 = new Float("-1e-30"); // f2 representará o float -1*10^-30 - a conversão de
20                                // string para float será feita automaticamente.
21     f3 = new Float(1e100);     // f3 representará o valor 1*10^100, que é maior do que
22                                // o maior valor representável por uma variável do tipo
23                                // float - neste caso, o compilador simplesmente
24                                // considerará f3 como representando o valor especial
25                                // Infinity (infinito).
26     f4 = new Float(-1./0.);    // f4 representará o valor -Infinity (infinito negativo,
27                                // resultado da divisão de um valor de ponto flutuante
28                                // por zero com um dos valores sendo negativo).
29     f5 = new Float(0./0.);     // f5 representará o valor NaN (Not a Number, não é um
30                                // número), que é o resultado da divisão de zero por
31                                // zero.
32     float f = f1.floatValue();   // recupera o valor encapsulado por f1 como um valor
33                                  // do tipo float
34     double d2 = f2.doubleValue(); // recupera o valor encapsulado por f2 como um valor
35                                   // do tipo double (aprox. -1e-30)
36     double d3 = f3.doubleValue(); // recupera o valor encapsulado por f3 como um valor
37                                   // do tipo double (Infinity)
38     // Compara dois Floats numericamente. Note que não podemos comparar estes valores
39     // diretamente com ==, !=  <, >=, etc.
40     if (f1.compareTo(f2) > 0) System.out.println(f1+" é maior que "+f2);
41     if (f3.compareTo(f2) > 0) System.out.println(f3+" é maior que "+f2);
42     // Podemos comparar instâncias da classe Float com valores constantes da classe
43     // Float usando métodos especiais.
44     if (f3.isInfinite()) System.out.println(f3+" é infinito.");
45     if (f5.isNaN()) System.out.println(f5+" não é um número.");
46     float trilhão = Float.parseFloat("1000000000000"); // decodifica uma string
47                                                        // contendo um valor do tipo
48                                                        // float, diretamente, sem
49                                                        // criar uma instância da
50                                                        // classe Float.
51     System.out.println("Faixa de valores do float:"+ // impressão dos valores-limite
52                        Float.MIN_VALUE+" a "+
53                        Float.MAX_VALUE);
54     }
55
56   } // fim da classe DemoClasseFloat
```

Nas linhas 16 e 19 da listagem A.29 temos a criação de duas instâncias da classe `Float`, uma com um valor de ponto flutuante e outro com uma string que representa um valor de ponto flutuante. A instância construída na linhas 21, 26 e 29 armazenarão valores especiais (apresentados na seção 13.2). As linhas 32, 34 e 36 mostram a conversão dos valores das instâncias para os tipos

`float` e `double`, e as linhas 40 a 45 demostram métodos de comparação. A linha 46 mostra outra maneira de inicialização de instâncias da classe `Float` através do método `valueOf`. As linhas 51 a 53 imprimem os valores mínimo e máximo para variáveis do tipo `float`.

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.30.

Listagem A.30: A classe `DemoClasseFloatComErros`, que demonstra usos incorretos da classe `Float`.

```
1  /**
2   * A classe DemoClasseFloatComErros, que demonstra vários usos incorretos e algumas
3   * armadilhas da classe Float.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoClasseFloatComErros // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias instâncias da classe Float, que encapsula um valor do tipo nativo float.
11   * O método também demonstra algumas operações com estas instâncias. Muitas das
12   * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
13   * compilado.
14   * @param argumentos os argumentos que podem ser passados para o método via linha
15   *        de comando, mas que neste caso serão ignorados.
16   */
17   public static void main(String[] argumentos)
18     {
19     Float f1,f2,f3,f4,f5,f6,f7,f8; // declaração das instâncias
20     f1 = new Double(1.0);          // ERRO: não é possível inicializar uma instância
21                                    //       da classe Float com uma instância da
22                                    //       classe Byte (ou de qualquer outra).
23     f2 = new Float(true);          // ERRO: não é possível converter do tipo boolean
24                                    //       para o tipo float, esperado pelo construtor
25                                    //       da classe Float.
26     f3 = 10e23f;                   // ERRO: valor do tipo nativo float não pode ser
27                                    //       convertido para instância da classe Float.
28     f4 = new Float("10000000000000000000000000000000000000000000000000");
29                                    // não é um erro, mas como o valor contido na
30                                    // string passada como argumento para o construtor
31                                    // contém um valor que não pode ser representado
32                                    // por um float, este será considerado Infinity.
33     f5 = new Float("355/113");     // ERRO: a string passada como argumento para o
34                                    //       construtor contém caracteres que não
35                                    //       permitem a conversão.
36     f6 = new Float();              // ERRO: não existe construtor vazio (que não
37                                    //       receba argumentos) para a classe Float
38     f7 = new Float(1e12);          // não existe erro, mas...
39     long l = f7.longValue();       // a variável l receberá o long mais próximo ao
40                                    // valor representado pela instância f7, que será
41                                    // 999999995904 - note a perda de precisão quando
42                                    // um valor relativamente grande do tipo float é
43                                    // convertido para um valor do tipo long;
44     int i = f7.intValue();         // a variável i valerá 2147483647 (o maior valor
45                                    // inteiro positivo representável);
46     short s = f7.shortValue();     // a variável s valerá -1;
47     byte b = f7.byteValue();       // e a variável b valerá -1.
48     f8 = new Float(1e90);          // não existe erro, mas como o valor do tipo double
49                                    // passado para o construtor da classe Float não
50                                    // pode ser representado por um float, este será
51                                    // considerado como sendo Infinity.
52     // Compara Floats com instâncias de outras classes, com o valor null e com
53     // valores de tipos nativos.
54     if (f7.equals(null)) System.out.println(f7+" é nulo."); // nada será impresso
55     if (f6.equals(-1)) System.out.println(f6+" vale -1.");  // ERRO: é impossível
56                                                            // comparar argumentos que
57                                                            // não sejam instâncias
58                                                            // com o método equals.
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
59    Double d = new Double(1e12); // criamos uma instância da classe Float
60    if (f7.equals(d))                          // nada será impresso, pois embora f7 e d
61      System.out.println(f7+" é igual a "+d); // sejam instâncias, são instâncias de
62                                               // classes diferentes, não podendo ser
63                                               // comparadas - o resultado será sempre
64                                               // false.
65    if (f7.compareTo(d) == 0)                  // ERRO: não será possível efetuar a
66      System.out.println(f7+" é igual a "+d); //       comparação entre classes
67    Float f9 = Float.valueOf("99999999999999999999999999999999999999999999999999999");
68                                               // não existe erro, mas como o valor
69                                               // contido na string não pode ser
70                                               // representado por um float, o valor
71                                               // encapsulado na classe será conside-
72                                               // rado como sendo Infinity.
73    Float f10 = Float.valueOf("ff");  // ERRO: o valor contido na string não pode ser
74                                      //       não pode ser convertido para float
75                                      //       (somente existe conversão com a base 10)
76    }
77
78  } // fim da classe DemoClasseFloatComErros
79
```

Nas linhas 20, 23, 26, 33 e 36 da listagem A.30 vemos o uso incorreto de construtores ou a passagem de argumentos que causarão erros de compilação ou de execução. As linhas 39 a 47 demonstram os problemas com perda de precisão e informação dos métodos de conversão. Outros erros de comparação e conversão também são comentados.

### A.3.8 A classe `Double`

*classe* `Double`

A classe `Double` encapsula valores do tipo nativo `double`. Os construtores da classe permitem a inicialização dos valores encapsulados usando como argumento um valor do tipo `double` (ou qualquer valor de qualquer tipo numérico, já que a conversão implícita para `double` pode ser feita sem problemas) ou de uma string que represente corretamente um valor do tipo `double`. Neste segundo caso, a string deve conter somente e exatamente os caracteres necessários para que um valor do tipo `double` seja representado.

Além dos construtores, outros métodos e campos úteis da classe `Double` são:

*campo* `MIN_VALUE`

- O campo estático `MIN_VALUE` representa o menor valor que pode ser armazenado em um campo do tipo `double`, e é igual a $4.9E-324$.

*campo* `MAX_VALUE`

- O campo estático `MAX_VALUE` representa o maior valor que pode ser armazenado em um campo do tipo `double`, e é igual a $1.7976931348623157E308$.

*campo* `NaN`

- O campo estático `NaN` representa o valor especial *Not a Number* do tipo `double`. Operações como divisão de zero por zero ou multiplicação de zero por infinito onde os argumentos são do tipo `double` terão como resultado este valor.

*campo* `POSITIVE_INFINITY`

- O campo estático `POSITIVE_INFINITY` representa o valor especial infinito positivo do tipo `double`. Divisões de valores positivos por zero onde os operadores são do tipo `double` terão como resultado este valor.

*campo* `NEGATIVE_INFINITY`

- O campo estático `NEGATIVE_INFINITY` representa o valor especial infinito negativo do tipo `double`. Divisões de valores negativos por zero onde os operadores são do tipo `double` terão como resultado este valor.

*método* `parseDouble`

- O método estático `parseDouble` recebe como argumento uma string que deve representar adequadamente um valor do tipo `double`, e retorna um valor do tipo `double` contendo o valor decodificado, ou lança a exceção `NumberFormatException` se a string conter espaços, caracteres não reconhecidos ou estiver formatada de uma maneira não reconhecível pelo método. Este método reconhece strings no formato *xey* como válidas, onde *x* é um

valor com ponto flutuante opcional, precedido de um sinal de menos também opcional, e *y* é um valor inteiro precedido opcionalmente de um sinal de menos.

- O método estático `valueOf` se comporta da mesma maneira que o método `parseDouble`, exceto que o método retorna uma instância da classe `Double` encapsulando o valor que foi passado como um argumento (instância da classe `String`). O método também causará a exceção `NumberFormatException` se a string não for reconhecida como um `double`. *método* `valueOf`
- O método `byteValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `byte`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `double` para um valor do tipo `byte`. *método* `byteValue`
- O método `shortValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `short`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `double` para um valor do tipo `short`. *método* `shortValue`
- O método `intValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `int`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `double` para um valor do tipo `int`. *método* `intValue`
- O método `longValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `long`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `double` para um valor do tipo `long`. *método* `longValue`
- O método `floatValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `float`. É importante notar que executar este método pode retornar um valor sem sentido por causa da perda de informação: esta conversão segue as mesmas regras do *cast* de um valor do tipo `double` para um valor do tipo `float`. *método* `floatValue`
- O método `doubleValue` retorna o valor encapsulado em uma instância desta classe como um valor do tipo nativo `double`. Este é o método comumente usado para "desencapsular" um valor encapsulado em uma instância da classe `Double`. *método* `doubleValue`
- O método `compareTo` compara a instância da classe `Double` com outra, que deve ser passada como um argumento para o método, retornando $-1$ se o valor encapsulado for menor que o valor passado como argumento, $+1$ se for maior e 0 se ambos forem iguais. Este método considera o valor `NaN` como sendo maior que todos os outros, até mesmo maior que infinito, e igual somente a si mesmo. *método* `compareTo`
- O método `equals` recebe como argumento uma instância de qualquer classe e retorna `true` se esta instância for da classe `Double` e o valor encapsulado pela instância passada como argumento for igual ao valor encapsulado. Uma única exceção: este método considera $+0$ e $-0$ (resultados da divisão por infinito, onde os sinais dos operadores determinam os sinais do zero) como sendo diferentes. *método* `equals`
- O método `isInfinite` tem duas formas: se chamado a partir de uma instância da classe `Double` ele retornará `true` se o valor encapsulado pela instância for infinito (positivo ou negativo). A outra forma é estática, recebe como argumento um valor do tipo `double` e retorna `true` se o valor passado como argumento for infinito. *método* `isInfinite`
- O método `isNaN` tem duas formas: se chamado a partir de uma instância da classe `Double` ele retornará `true` se o valor encapsulado pela instância for `NaN`. A outra forma é estática, recebe como argumento um valor do tipo `double` e retorna `true` se o valor passado como argumento for `NaN`. *método* `isNaN`



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

*método* `toString`

- O método `toString` retorna uma string contendo o valor encapsulado. Este método pode ser chamado implicitamente, da mesma forma que outros métodos `toString` de outras classes. Este método tem uma outra formas (que não pode ser chamada implicitamente) que é declaradas como `static`, uma recebe como argumento um valor do tipo `double` e retorna uma string representando aquele valor.

Alguns exemplos de uso de instâncias da classe `Double` são mostrados no programa na listagem A.31.

Listagem A.31: A classe `DemoClasseDouble`, que demonstra usos da classe `Double`.

```
1  /**
2   * A classe DemoClasseDouble, que demonstra vários usos corretos da classe Double.
3   */
4  class DemoClasseDouble // declaração da classe
5    {
6   /**
7    * O método main permite a execução desta classe. Este método contém declarações de
8    * várias instâncias da classe Double, que encapsula um valor do tipo nativo double.
9    * O método também demonstra algumas operações com estas instâncias.
10   * @param argumentos os argumentos que podem ser passados para o método via linha
11   *        de comando, mas que neste caso serão ignorados.
12   */
13   public static void main(String[] argumentos)
14     {
15     Double d1,d2,d3,d4,d5;      // declaração das instâncias
16     d1 = new Double(1e300);     // d1 representará o double 1*10^300.
17     d2 = new Double("1e-300"); // f2 representará o double 1*10^-300 - a conversão de
18                                // string para double será feita automaticamente.
19     d3 = new Double(1./0.);     // d3 representará o valor Infinity (infinito positivo,
20                                // resultado da divisão de um valor de ponto flutuante
21                                // por zero).
22     d4 = new Double(-1./0.);    // d4 representará o valor -Infinity (infinito negativo,
23                                // resultado da divisão de um valor de ponto flutuante
24                                // por zero, com um dos dois valores sendo negativo).
25     d5 = new Double(0./0.);     // d5 representará o valor NaN (Not a Number, não é um
26                                // número), que é o resultado da divisão de zero por
27                                // zero.
28     double db1 = d1.doubleValue(); // recupera o valor encapsulado por d1 como um valor
29                                    // do tipo double (1.0e300)
30     double db5 = d5.doubleValue(); // recupera o valor encapsulado por f2 como um valor
31                                    // do tipo double (NaN)
32     // Compara dois Doubles numericamente. Note que não podemos comparar estes valores
33     // diretamente com ==, !=  <, >=, etc.
34     if (d1.compareTo(d2) > 0) System.out.println(d1+" é maior que "+d2);
35     if (d4.compareTo(d3) < 0) System.out.println(d4+" é menor que "+d3);
36     // Podemos comparar instâncias da classe Double com valores constantes da classe
37     // Double usando métodos especiais.
38     if (d3.isInfinite()) System.out.println(d3+" é infinito.");
39     if (d5.isNaN()) System.out.println(d5+" não é um número.");
40     double zilhão = Double.parseDouble("1e200"); // decodifica uma string contendo
41                                                  // um valor do tipo double,
42                                                  // diretamente, sem criar uma
43                                                  // instância da classe Double
44     System.out.println("Faixa de valores do double:"+ // impressão dos valores-limite
45                        Double.MIN_VALUE+" a "+
46                        Double.MAX_VALUE);
47     }
48
49   } // fim da classe DemoClasseDouble
```

Na linhas 16 e 17 da listagem A.31 temos a inicialização de duas instâncias da classe `Double`, uma com um valor do tipo `double` e outra com uma string. Nas linhas 19, 22 e 25 temos inicialização de outras instâncias com os valores numéricos especiais (que foram descritos na



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

seção 13.2). Nas linhas 28 e 30 temos a recuperação dos valores das instâncias como valores do tipo `double`.

As linhas 34 a 39 demonstram os métodos de comparação desta classe. A linha 40 demonstra um método de conversão de string para `Double`, e as linhas 44 a 46 imprimirão os valores mínimo e máximo que podem ser representados por valores do tipo `double`.

Alguns exemplos de uso incorreto da classe são mostrados no programa na listagem A.32.

Listagem A.32: A classe `DemoClasseDoubleComErros`, que demonstra usos incorretos da classe `Double`.

```
1  /**
2   * A classe DemoClasseDoubleComErros, que demonstra vários usos incorretos e algumas
3   * armadilhas da classe Double.
4   * ESTA CLASSE NÃO PODE SER COMPILADA POR CAUSA DE ERROS INTENCIONAIS.
5   */
6  class DemoClasseDoubleComErros // declaração da classe
7    {
8   /**
9    * O método main permite a execução desta classe. Este método contém declarações de
10   * várias instâncias da classe Double, que encapsula um valor do tipo nativo double.
11   * O método também demonstra algumas operações com estas instâncias. Muitas das
12   * declarações e chamadas aos métodos estão incorretas, impedindo o programa de ser
13   * compilado.
14   * @param argumentos os argumentos que podem ser passados para o método via linha
15   *        de comando, mas que neste caso serão ignorados.
16   */
17    public static void main(String[] argumentos)
18      {
19      Double d1,d2,d3,d4,d5,d6,d7;     // declaração das instâncias
20      d1 = new Float(1.0);             // ERRO: não é possível inicializar uma instância
21                                       //       da classe Double com uma instância da
22                                       //       classe Float (ou de qualquer outra).
23      d2 = new Double(true);           // ERRO: não é possível converter do tipo boolean
24                                       //       para o tipo double, esperado pelo
25                                       //       construtor da classe Double.
26      d3 = 10e200;                     // ERRO: valor do tipo nativo double não pode ser
27                                       //       convertido para instância da classe
28                                       //       Double.
29      d4 = new Double("1000000000000000000000000000000000000000000000000000000000000000000000000"+
30                      "0000000000000000000000000000000000000000000000000000000000000000000000000"+
31                      "0000000000000000000000000000000000000000000000000000000000000000000000000"+
32                      "0000000000000000000000000000000000000000000000000000000000000000000000000"+
33                      "0000000000000000000000000000000000000000000000000000000000000000000000000");
34                                       // não é um erro, mas como o valor contido na
35                                       // string passada como argumento para o construtor
36                                       // contém um valor que não pode ser representado
37                                       // por um double, este será considerado Infinity.
38                                       // Note que a string passada para o construtor
39                                       // está sendo concatenada e não somada - o código
40                                       // foi escrito assim para facilitar a sua leitura.
41      d5 = new Double("1/7");          // ERRO: a string passada como argumento para o
42                                       //       construtor contém caracteres que não
43                                       //       permitem a conversão.
44      d6 = new Double();               // ERRO: não existe construtor vazio (que não
45                                       //       receba argumentos) para a classe Double
46      d7 = new Double(-1e109);         // não existe erro, mas...
47      float f = d7.floatValue();       // a variável f valerá -Infinity pois valores do
48                                       // tipo float não conseguem representar o valor
49                                       // encapsulado na instância d7 - Infinity negativo
50                                       // é o mais próximo.
51      long l = d7.longValue();         // a variável l receberá o long mais próximo ao
52                                       // valor representado pela instância d7, que será
53                                       // -9223372036854775808.
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
54      int i = d7.intValue();          // a variável i valerá -21474836478 (o maior valor
55                                      // inteiro negativo representável);
56      short s = d7.shortValue();      // a variável s valerá 0;
57      byte b = d7.byteValue();        // e a variável b valerá 0.
58      // Compara Doubles com instâncias de outras classes, com o valor null e com
59      // valores de tipos nativos.
60      if (d7.equals(null)) System.out.println(d7+" é nulo."); // nada será impresso
61      if (d6.equals(0)) System.out.println(d6+" vale 0.");  // ERRO: é impossível
62                                                            // comparar argumentos que
63                                                            // não sejam instâncias com
64                                                            // o método equals.
65      Float f = new Float(1./7.); // criamos uma instância da classe Float
66      if (d5.equals(f))                       // nada será impresso, pois embora d5 e f
67        System.out.println(d5+" é igual a "+f); // sejam instâncias, são instâncias de
68                                              // classes diferentes, não podendo ser
69                                              // comparadas - o resultado será sempre
70                                              // false.
71      if (d5.compareTo(f) == 0)               // ERRO: não será possível efetuar a
72        System.out.println(d7+" é igual a "+d); //       comparação entre classes
73      Double d8 = Double.valueOf("99999999999999999999999999999999999999999999999999999"+
74                                 "99999999999999999999999999999999999999999999999999999"+
75                                 "99999999999999999999999999999999999999999999999999999"+
76                                 "99999999999999999999999999999999999999999999999999999"+
77                                 "99999999999999999999999999999999999999999999999999999"+
78                                 "99999999999999999999999999999999999999999999999999999"+
79                                 "999999999");  // não existe erro, mas como o valor
80                                                // contido na string não pode ser
81                                                // representado por um double, o valor
82                                                // encapsulado na classe será conside-
83                                                // rado como sendo Infinity.
84      Double d9 = Double.valueOf("0.00000000000000000000000000000000000000000000000000000"+
85                                 "00000000000000000000000000000000000000000000000000000"+
86                                 "00000000000000000000000000000000000000000000000000000"+
87                                 "00000000000000000000000000000000000000000000000000000"+
88                                 "00000000000000000000000000000000000000000000000000000"+
89                                 "00000000000000000000000000000000000000000000000000000"+
90                                 "000000000000000001"); // não existe erro, mas como o
91                                                        // valor contido na string não
92                                                        // pode ser representado por um
93                                                        // double, o valor encapsulado
94                                                        // na classe será considerado
95                                                        // como sendo zero.
96      Double d10 = Double.valueOf("f"); // ERRO: o valor contido na string não pode ser
97                                        //       não pode ser convertido para double
98                                        //       (somente existe conversão com a base 10)
99      }
100
101   } // fim da classe DemoClasseDoubleComErros
```

As linhas 20, 23, 26, 41 e 44 da listagem A.32 demonstram chamadas incorretas a construtores, que causarão erros de compilação ou execução. As linhas 47 a 57 demonstram os erros de perda de precisão ou informação na conversão do valor encapsulado em uma instância da classe `Double` para tipos nativos. As linhas 60 a 72 demonstram erros de comparação e as linhas 73 a 96 demonstram erros de conversão de strings para instâncias da classe `Double`.

# Apêndice B

# A Classe `Keyboard`

## B.1 Introdução

A linguagem Java não tem classes ou métodos que permitam a entrada direta de dados via teclado para aplicações que não usem interfaces gráficas, o que diminui a possibilidade de interação entre o usuário e uma aplicação. Uma solução comum é usar a instância da classe `InputStream System.in`, que está sempre aberta e pode ser usada para a leitura de bytes ou linhas de texto como strings.

*A classe* `Keyboard`

Uma solução melhor é apresentada neste apêndice: a classe `Keyboard`, que encapsula o processo de leitura de strings de `System.in`, juntamente com rotinas de conversão de strings para dados nativos e mecanismos de uso de valores *default*. O código-fonte da classe `Keyboard` e exemplos de aplicação são mostrados neste apêndice.

Para usar a classe `Keyboard` em programas e classes, basta que ela esteja em algum diretório que contenha classes e que seja usado pelo compilador e interpretador Java. Para maiores informações sobre estes diretórios, veja o capítulo . Para aplicações simples, basta copiar a classe `Keyboard` para o diretório contendo os programas e classes sendo desenvolvidos.

O livro *An Introduction to Computer Science Using Java*, de Samuel N. Kamin, M. Dennis Mickunas e Edward M. Reingold, da editora McGraw-Hill, ISBN 0-07-03422405 também apresenta uma classe para encapsular entrada de dados via teclado, também chamada `Keyboard`. A classe `Keyboard` usada neste livro foi baseada na classe `EasyIn`, de Peter van der Linden, com muitas modificações, descritas no código na forma de comentários.

## B.2 A classe `Keyboard`

*Código-fonte da classe* `Keyboard`

A classe `Keyboard` é uma classe onde todos os métodos são estáticos, podendo ser usados diretamente, sem necessidade da criação de instâncias da classe. O código-fonte completo da classe `Keyboard` é mostrado na listagem B.1.

Listagem B.1: A classe `Keyboard`, que permite entrada simples via teclado.

```
1  /* As seguintes linhas permitem o uso das classes de entrada e saída e de processamento
2     de tokens */
3  import java.io.*;
4  import java.util.*;
5
6  /**
7   * A classe Keyboard, que permite a leitura de dados de tipos nativos e da classe
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
 * String a partir do teclado. Basicamente esta classe encapsula o funcionamento do
 * Stream System.in, que pode ser usado como entrada padrão de dados.
 *
 * Esta classe é baseada na classe EasyIn, de Peter van der Linden. As principais
 * modificações foram:
 * - Os métodos são estáticos para facilitar o uso da classe
 * - Criação de métodos sobrecarregados para que valores default possam ser usados
 * - Melhor tratamento das exceções
 * - Reconhecimento de vários valores como booleanos
 * - Descrição dos detalhes de implementação com comentários
 *
 * A mensagem original de copyright de Peter van de Linden é mostrada ao final desta
 * listagem. Vale a pena notar que a sintaxe proposta e mostrada na mensagem é
 * diferente da usada pela classe Keyboard.
 *
 * O livro "An Introduction to Computer Science Using Java", de Samuel N. Kamin,
 * M. Dennis Mickunas e Edward M. Reingold (editora McGraw-Hill, ISBN 0-07-034224-5)
 * também apresenta uma classe Keyboard, cujo funcionamento e código são diferentes
 * desta.
 *
 * @author Rafael Santos
 */

class Keyboard
  {
 /**
  * Declaração dos campos privados desta classe.
  */
  // O campo is é uma instância da classe InputStreamReader que será construída a
  // partir do stream System.in. Note que já que esta classe só tem métodos estáticos,
  // não terá construtor, mas podemos inicializar instâncias de classes desta forma.
  private static InputStreamReader is = new InputStreamReader( System.in );
  // O campo br é uma instância da classe BufferedReader que usa is como argumento
  // para seu construtor.
  private static BufferedReader br = new BufferedReader( is );
  // O campo st é uma instância da classe StringTokenizer que será usada em vários
  // métodos na classe.
  private static StringTokenizer st;
  // O campo nt é uma instância da classe String que será usada em vários métodos
  // na classe.
  private static String nt;
  // O campo debug estabelece se mensagens de erro de conversão e leitura devem
  // ser mostradas ou não. O campo não pode ser modificado diretamente, mas
  // através de métodos específicos descritos abaixo.
  static boolean debug = false;

 /**
  * O método getToken lê uma String do BufferedReader associado com a entrada padrão
  * e retorna uma instância de StringTokenizer contendo os tokens criados a partir
  * da linha lida. O método é declarado como private pois só deverá ser chamado
  * a partir de outros métodos da classe.
  * @return uma instância de StringTokenizer contendo os tokens obtidos da linha lida
  * @exception IOException se um erro de entrada e saída ocorrer.
  */
  private static StringTokenizer getToken() throws IOException
    {
    String s = br.readLine();
    return new StringTokenizer(s);
    } // fim do método getToken

 /**
  * O método readBoolean lê e retorna um valor do tipo boolean. Este método
  * simplesmente chama o método readBoolean com argumentos, descrito abaixo,
  * considerando o valor default como sendo true.
  * @return o valor do tipo boolean lido (ou, em caso de erro de leitura, true)
  */
  public static boolean readBoolean()
    {
    return readBoolean(true);
```

```
77      }
78
79    /**
80     * O método readBoolean lê e retorna um valor do tipo boolean. Este método tenta
81     * pegar o próximo token a ser lido da entrada padrão e o compara com algumas
82     * Strings constantes para verificar a igualdade, e retorna true ou false dependendo
83     * do resultado da comparação.
84     * @param defaultvalue o valor default caso não seja possível ler e decodificar
85     *                     um valor booleano
86     * @return o valor do tipo boolean lido (ou, em caso de erro de leitura, igual ao
87     *         argumento passado para o método
88     */
89    public static boolean readBoolean(boolean defaultvalue)
90      {
91      try
92        {
93        st = getToken(); // pega os tokens a partir da linha lida
94        nt = st.nextToken(); // e a primeira String do token.
95        // Retorna true se a String for igual a...
96        if (nt.equalsIgnoreCase("true") ||        // true
97            nt.equalsIgnoreCase("t") ||           // t
98            nt.equalsIgnoreCase("yes") ||         // yes
99            nt.equalsIgnoreCase("y") ||           // y
100           nt.equalsIgnoreCase("on") ||          // on
101           nt.equalsIgnoreCase("v") ||           // v
102           nt.equalsIgnoreCase("s") ||           // s
103           nt.equalsIgnoreCase("sim") ||         // sim
104           nt.equalsIgnoreCase("verdadeiro"))    // verdadeiro
105           return true;
106       // Retorna false se a String for igual a...
107       else if (nt.equalsIgnoreCase("false") ||  // false
108                nt.equalsIgnoreCase("f") ||      // f
109                nt.equalsIgnoreCase("no") ||     // no
110                nt.equalsIgnoreCase("n") ||      // n
111                nt.equalsIgnoreCase("off") ||    // off
112                nt.equalsIgnoreCase("nao") ||    // nao
113                nt.equalsIgnoreCase("não") ||    // não
114                nt.equalsIgnoreCase("falso"))    // falso
115           return false;
116       // Se não for nenhum dos valores reconhecidos, retorna o default
117       else return defaultvalue;
118       }
119     catch (IOException ioe) // se houver algum erro de leitura
120       {
121       if (debug) // se for pedida a impressão de mensagens de erro
122         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um boolean. "+
123                            "Retorna "+defaultvalue);
124       return defaultvalue; // retorna o valor default
125       }
126     catch (NoSuchElementException nsee) // se não houverem tokens
127       {
128       if (debug) // se for pedida a impressão de mensagens de erro
129         System.err.println("KEYBOARD:: Entrada não contém um boolean. "+
130                            "Retorna "+defaultvalue);
131       return defaultvalue; // retorna o valor default
132       }
133     } // fim do método readBoolean
134
135   /**
136    * O método readByte lê e retorna um valor do tipo byte. Este método
137    * simplesmente chama o método readByte com argumentos, descrito abaixo,
138    * considerando o valor default como sendo (byte)0.
139    * @return o valor do tipo byte lido (ou, em caso de erro de leitura, (byte)0)
140    */
141   public static byte readByte()
142     {
143     return readByte((byte)0);
144     } // fim do método readByte
145
```

```
146  /**
147   * O método readByte lê e retorna um valor do tipo byte. Este método tenta
148   * pegar o próximo token a ser lido da entrada padrão e o passa como argumento para
149   * o construtor da classe Byte, que tenta convertê-lo. Se a leitura e conversão
150   * puderem ser feitas, um valor do tipo byte é retornado, caso contrário o valor
151   * default (passado como argumento) é retornado.
152   * @param defaultvalue o valor default caso não seja possível ler e decodificar
153   *                     um valor do tipo byte
154   * @return o valor do tipo byte lido (ou, em caso de erro de leitura, igual ao
155   *         argumento passado para o método
156   */
157   public static byte readByte(byte defaultvalue)
158     {
159     try
160       {
161       st = getToken(); // pega os tokens a partir da linha lida
162       nt = st.nextToken(); // e a primeira String do token.
163       // retorna o valor processado pela classe Byte
164       return Byte.parseByte(nt);
165       }
166     catch (IOException ioe) // se houver algum erro de leitura
167       {
168       if (debug) // se for pedida a impressão de mensagens de erro
169         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um byte. "+
170                            "Retorna "+defaultvalue);
171       return defaultvalue; // retorna o valor default
172       }
173     catch (NumberFormatException nfe) // se houver algum erro de conversão
174       {
175       if (debug) // se for pedida a impressão de mensagens de erro
176         System.err.println("KEYBOARD:: Erro de conversão de "+nt+" para um byte. "+
177                            "Retorna "+defaultvalue);
178       return defaultvalue;
179       }
180     catch (NoSuchElementException nsee) // se não houverem tokens
181       {
182       if (debug) // se for pedida a impressão de mensagens de erro
183         System.err.println("KEYBOARD:: Entrada não contém um byte. "+
184                            "Retorna "+defaultvalue);
185       return defaultvalue; // retorna o valor default
186       }
187     } // fim do método readByte
188
189  /**
190   * O método readShort lê e retorna um valor do tipo short. Este método
191   * simplesmente chama o método readShort com argumentos, descrito abaixo,
192   * considerando o valor default como sendo (short)0.
193   * @return o valor do tipo short lido (ou, em caso de erro de leitura, (short)0)
194   */
195   public static short readShort()
196     {
197     return readShort((short)0);
198     } // fim do método readShort
199
200  /**
201   * O método readShort lê e retorna um valor do tipo short. Este método tenta
202   * pegar o próximo token a ser lido da entrada padrão e o passa como argumento para
203   * o construtor da classe Short, que tenta convertê-lo. Se a leitura e conversão
204   * puderem ser feitas, um valor do tipo short é retornado, caso contrário o valor
205   * default (passado como argumento) é retornado.
206   * @param defaultvalue o valor default caso não seja possível ler e decodificar
207   *                     um valor do tipo short
208   * @return o valor do tipo short lido (ou, em caso de erro de leitura, igual ao
209   *         argumento passado para o método
210   */
211   public static short readShort(short defaultvalue)
212     {
213     try
214       {
```

```
215       st = getToken(); // pega os tokens a partir da linha lida
216       nt = st.nextToken(); // e a primeira String do token.
217       // retorna o valor processado pela classe Short
218       return Short.parseShort(nt);
219       }
220     catch (IOException ioe) // se houver algum erro de leitura
221       {
222       if (debug) // se for pedida a impressão de mensagens de erro
223         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um short. "+
224                            "Retorna "+defaultvalue);
225       return defaultvalue; // retorna o valor default
226       }
227     catch (NumberFormatException nfe) // se houver algum erro de conversão
228       {
229       if (debug) // se for pedida a impressão de mensagens de erro
230         System.err.println("KEYBOARD:: Erro de conversão de "+nt+" para um short. "+
231                            "Retorna "+defaultvalue);
232       return defaultvalue;
233       }
234     catch (NoSuchElementException nsee) // se não houverem tokens
235       {
236       if (debug) // se for pedida a impressão de mensagens de erro
237         System.err.println("KEYBOARD:: Entrada não contém um short. "+
238                            "Retorna "+defaultvalue);
239       return defaultvalue; // retorna o valor default
240       }
241     } // fim do método readShort
242
243   /**
244    * O método readInt lê e retorna um valor do tipo int. Este método
245    * simplesmente chama o método readInt com argumentos, descrito abaixo,
246    * considerando o valor default como sendo 0.
247    * @return o valor do tipo int lido (ou, em caso de erro de leitura, 0)
248    */
249    public static int readInt()
250      {
251      return readInt(0);
252      } // fim do método readInt
253
254   /**
255    * O método readInt lê e retorna um valor do tipo int. Este método tenta
256    * pegar o próximo token a ser lido da entrada padrão e o passa como argumento para
257    * o construtor da classe Int, que tenta convertê-lo. Se a leitura e conversão
258    * puderem ser feitas, um valor do tipo int é retornado, caso contrário o valor
259    * default (passado como argumento) é retornado.
260    * @param defaultvalue o valor default caso não seja possível ler e decodificar
261    *                     um valor do tipo int
262    * @return o valor do tipo int lido (ou, em caso de erro de leitura, igual ao
263    *         argumento passado para o método
264    */
265    public static int readInt(int defaultvalue)
266      {
267      try
268        {
269        st = getToken(); // pega os tokens a partir da linha lida
270        nt = st.nextToken(); // e a primeira String do token.
271        // retorna o valor processado pela classe Integer
272        return Integer.parseInt(nt);
273        }
274      catch (IOException ioe) // se houver algum erro de leitura
275        {
276        if (debug) // se for pedida a impressão de mensagens de erro
277          System.err.println("KEYBOARD:: Erro de entrada e saída lendo um int. "+
278                             "Retorna "+defaultvalue);
279        return defaultvalue; // retorna o valor default
280        }
281      catch (NumberFormatException nfe) // se houver algum erro de conversão
282        {
283        if (debug) // se for pedida a impressão de mensagens de erro
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
284         System.err.println("KEYBOARD:: Erro de conversão de "+nt+" para um int. "+
285                            "Retorna "+defaultvalue);
286       return defaultvalue;
287       }
288     catch (NoSuchElementException nsee) // se não houverem tokens
289       {
290       if (debug) // se for pedida a impressão de mensagens de erro
291         System.err.println("KEYBOARD:: Entrada não contém um int. "+
292                            "Retorna "+defaultvalue);
293       return defaultvalue; // retorna o valor default
294       }
295     } // fim do método readInt
296
297   /**
298    * O método readLong lê e retorna um valor do tipo long. Este método
299    * simplesmente chama o método readLong com argumentos, descrito abaixo,
300    * considerando o valor default como sendo 0l.
301    * @return o valor do tipo long lido (ou, em caso de erro de leitura, 0l)
302    */
303   public static long readLong()
304     {
305     return readLong(0l);
306     } // fim do método readLong
307
308   /**
309    * O método readLong lê e retorna um valor do tipo long. Este método tenta
310    * pegar o próximo token a ser lido da entrada padrão e o passa como argumento para
311    * o construtor da classe Long, que tenta convertê-lo. Se a leitura e conversão
312    * puderem ser feitas, um valor do tipo long é retornado, caso contrário o valor
313    * default (passado como argumento) é retornado.
314    * @param defaultvalue o valor default caso não seja possível ler e decodificar
315    *                     um valor do tipo long
316    * @return o valor do tipo long lido (ou, em caso de erro de leitura, igual ao
317    *         argumento passado para o método
318    */
319   public static long readLong(long defaultvalue)
320     {
321     try
322       {
323       st = getToken(); // pega os tokens a partir da linha lida
324       nt = st.nextToken(); // e a primeira String do token.
325       // retorna o valor processado pela classe Long
326       return Long.parseLong(nt);
327       }
328     catch (IOException ioe) // se houver algum erro de leitura
329       {
330       if (debug) // se for pedida a impressão de mensagens de erro
331         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um long. "+
332                            "Retorna "+defaultvalue);
333       return defaultvalue; // retorna o valor default
334       }
335     catch (NumberFormatException nfe) // se houver algum erro de conversão
336       {
337       if (debug) // se for pedida a impressão de mensagens de erro
338         System.err.println("KEYBOARD:: Erro de conversão de "+nt+" para um long. "+
339                            "Retorna "+defaultvalue);
340       return defaultvalue;
341       }
342     catch (NoSuchElementException nsee) // se não houverem tokens
343       {
344       if (debug) // se for pedida a impressão de mensagens de erro
345         System.err.println("KEYBOARD:: Entrada não contém um long. "+
346                            "Retorna "+defaultvalue);
347       return defaultvalue; // retorna o valor default
348       }
349     } // fim do método readLong
350
351   /**
352    * O método readFloat lê e retorna um valor do tipo float. Este método
```

```
353   * simplesmente chama o método readFloat com argumentos, descrito abaixo,
354   * considerando o valor default como sendo 0f.
355   * @return o valor do tipo float lido (ou, em caso de erro de leitura, 0f)
356   */
357   public static float readFloat()
358     {
359     return readFloat(0f);
360     } // fim do método readFloat
361
362  /**
363   * O método readFloat lê e retorna um valor do tipo float. Este método tenta
364   * pegar o próximo token a ser lido da entrada padrão e o passa como argumento para
365   * o construtor da classe Float, que tenta convertê-lo. Se a leitura e conversão
366   * puderem ser feitas, um valor do tipo float é retornado, caso contrário o valor
367   * default (passado como argumento) é retornado.
368   * @param defaultvalue o valor default caso não seja possível ler e decodificar
369   *                     um valor do tipo float
370   * @return o valor do tipo float lido (ou, em caso de erro de leitura, igual ao
371   *         argumento passado para o método
372   */
373   public static float readFloat(float defaultvalue)
374     {
375     try
376       {
377       st = getToken(); // pega os tokens a partir da linha lida
378       nt = st.nextToken(); // e a primeira String do token.
379       // retorna o valor processado pela classe Float
380       return Float.parseFloat(nt);
381       }
382     catch (IOException ioe) // se houver algum erro de leitura
383       {
384       if (debug) // se for pedida a impressão de mensagens de erro
385         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um float. "+
386                            "Retorna "+defaultvalue);
387       return defaultvalue; // retorna o valor default
388       }
389     catch (NumberFormatException nfe) // se houver algum erro de conversão
390       {
391       if (debug) // se for pedida a impressão de mensagens de erro
392         System.err.println("KEYBOARD:: Erro de conversão de "+nt+" para um float. "+
393                            "Retorna "+defaultvalue);
394       return defaultvalue;
395       }
396     catch (NoSuchElementException nsee) // se não houverem tokens
397       {
398       if (debug) // se for pedida a impressão de mensagens de erro
399         System.err.println("KEYBOARD:: Entrada não contém um float. "+
400                            "Retorna "+defaultvalue);
401       return defaultvalue; // retorna o valor default
402       }
403     } // fim do método readFloat
404
405  /**
406   * O método readDouble lê e retorna um valor do tipo double. Este método
407   * simplesmente chama o método readDouble com argumentos, descrito abaixo,
408   * considerando o valor default como sendo 0.0.
409   * @return o valor do tipo double lido (ou, em caso de erro de leitura, 0.0)
410   */
411   public static double readDouble()
412     {
413     return readDouble(0.0);
414     } // fim do método readDouble
415
416  /**
417   * O método readDouble lê e retorna um valor do tipo double. Este método tenta
418   * pegar o próximo token a ser lido da entrada padrão e o passa como argumento para
419   * o construtor da classe Double, que tenta convertê-lo. Se a leitura e conversão
420   * puderem ser feitas, um valor do tipo double é retornado, caso contrário o valor
421   * default (passado como argumento) é retornado.
```



Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

```
422   * @param defaultvalue o valor default caso não seja possível ler e decodificar
423   *                      um valor do tipo double
424   * @return o valor do tipo double lido (ou, em caso de erro de leitura, igual ao
425   *         argumento passado para o método
426   */
427   public static double readDouble(double defaultvalue)
428     {
429     try
430       {
431       st = getToken(); // pega os tokens a partir da linha lida
432       nt = st.nextToken(); // e a primeira String do token.
433       // retorna o valor processado pela classe Double
434       return Double.parseDouble(nt);
435       }
436     catch (IOException ioe) // se houver algum erro de leitura
437       {
438       if (debug) // se for pedida a impressão de mensagens de erro
439         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um double. "+
440                            "Retorna "+defaultvalue);
441       return defaultvalue; // retorna o valor default
442       }
443     catch (NumberFormatException nfe) // se houver algum erro de conversão
444       {
445       if (debug) // se for pedida a impressão de mensagens de erro
446         System.err.println("KEYBOARD:: Erro de conversão de "+nt+" para um double. "+
447                            "Retorna "+defaultvalue);
448       return defaultvalue;
449       }
450     catch (NoSuchElementException nsee) // se não houverem tokens
451       {
452       if (debug) // se for pedida a impressão de mensagens de erro
453         System.err.println("KEYBOARD:: Entrada não contém um double. "+
454                            "Retorna "+defaultvalue);
455       return defaultvalue; // retorna o valor default
456       }
457     } // fim do método readDouble
458
459  /**
460   * O método readChar lê e retorna um valor do tipo char. Este método
461   * simplesmente chama o método readChar com argumentos, descrito abaixo,
462   * considerando o valor default como sendo ' ' (espaço).
463   * @return o valor do tipo char lido (ou, em caso de erro de leitura, espaço)
464   */
465   public static char readChar()
466     {
467     return readChar(' ');
468     } // fim do método readChar
469
470  /**
471   * O método readChar lê e retorna um valor do tipo char. Este método tenta
472   * pegar o próximo token a ser lido da entrada padrão e retorna o primeiro caracter
473   * deste token, ignorando os outros. Se a leitura puder ser feita, um valor do tipo
474   * char é retornado, caso contrário o valor default (passado como argumento) é
475   * retornado.
476   * @param defaultvalue o valor default caso não seja possível ler um valor do tipo
477   *                     char
478   * @return o valor do tipo char lido (ou, em caso de erro de leitura, igual ao
479   *         argumento passado para o método
480   */
481   public static char readChar(char defaultvalue)
482     {
483     try
484       {
485       st = getToken(); // pega os tokens a partir da linha lida
486       nt = st.nextToken(); // e a primeira String do token.
487       // retorna o primeiro caracter da String acima
488       return nt.charAt(0);
489       }
490     catch (IOException ioe) // se houver algum erro de leitura
```

```
491       {
492       if (debug) // se for pedida a impressão de mensagens de erro
493         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um char. "+
494                            "Retorna "+defaultvalue);
495       return defaultvalue; // retorna o valor default
496       }
497     catch (NoSuchElementException nsee) // se não houverem tokens
498       {
499       if (debug) // se for pedida a impressão de mensagens de erro
500         System.err.println("KEYBOARD:: Entrada não contém um char. "+
501                            "Retorna "+defaultvalue);
502       return defaultvalue; // retorna o valor default
503       }
504     } // fim do método readChar
505
506   /**
507    * O método readString lê e retorna uma instância da classe String. Este método
508    * simplesmente chama o método readString com argumentos, descrito abaixo,
509    * considerando o valor default como sendo "" (String vazia).
510    * @return o valor do tipo String lido (ou, em caso de erro de leitura, uma String
511    *         vazia)
512    */
513   public static String readString()
514     {
515     return readString("");
516     } // fim do método readString
517
518   /**
519    * O método readString lê e retorna um valor do tipo String. Este método retorna
520    * uma linha inteira lida da entrada padrão, sem processá-la como tokens. Se a
521    * leitura puder ser feita, a String lida é retornada, caso contrário o valor
522    * default (passado como argumento) é retornado. O valor default também é retornado
523    * no caso de uma String vazia.
524    * @param defaultvalue o valor default caso não seja possível ler um valor do tipo
525    *                     String
526    * @return o valor do tipo String lido (ou, em caso de erro de leitura, igual ao
527    *         argumento passado para o método
528    */
529   public static String readString(String defaultvalue)
530     {
531     try
532       {
533       nt = br.readLine(); // uma String lida diretamente
534       if (nt.trim().length() == 0) // nada foi entrado, então
535         return defaultvalue; // retorna o valor default
536       else return nt; // retorna o que foi lido
537       }
538     catch (IOException ioe) // se houver algum erro de leitura
539       {
540       if (debug) // se for pedida a impressão de mensagens de erro
541         System.err.println("KEYBOARD:: Erro de entrada e saída lendo um String. "+
542                            "Retorna "+defaultvalue);
543       return defaultvalue; // retorna o valor default
544       }
545     } // fim do método readString
546
547   /**
548    * O método debugOn modifica o campo que indica que mensagens de erro deverão ser
549    * mostradas, fazendo com que sejam mostradas até que o método debugOff seja chamado.
550    */
551   public static void debugOn()
552     {
553     debug = true;
554     System.err.println("KEYBOARD:: Mostrando mensagens de erro e avisos...");
555     } // fim do método debugOn
556
557   /**
558    * O método debugOff modifica o campo que indica que mensagens de erro deverão ser
559    * mostradas, fazendo com que não sejam mostradas até que o método debufOn seja
```

```
560   * chamado.
561   */
562   public static void debugOff()
563     {
564     debug = false;
565     } // fim do método debugOff
566
567   } // fim da classe Keyboard
568
569 // Segue a mensagem original da classe EasyIn de Peter van der Linden
570
571 // Simple input from the keyboard for all primitive types. ver 1.0
572 // Copyright (c) Peter van der Linden,  May 5 1997.
573 //     corrected error message 11/21/97
574 //
575 // The creator of this software hereby gives you permission to:
576 //  1. copy the work without changing it
577 //  2. modify the work providing you send me a copy which I can
578 //     use in any way I want, including incorporating into this work.
579 //  3. distribute copies of the work to the public by sale, lease,
580 //     rental, or lending
581 //  4. perform the work
582 //  5. display the work
583 //  6. fold the work into a funny hat and wear it on your head.
584 //
585 // This is not thread safe, not high performance, and doesn't tell EOF.
586 // It's intended for low-volume easy keyboard input.
587 // An example of use is:
588 //     EasyIn easy = new EasyIn();
589 //     int i = easy.readInt();   // reads an int from System.in
590 //     float f = easy.readFloat();   // reads a float from System.in
```

### B.2.1 Exemplos de uso métodos da classe `Keyboard`

*Exemplos de uso da classe* `Keyboard`

Como os métodos da classe `Keyboard` são todos estáticos, para usá-los não é necessário instanciar a classe. Os métodos para leitura de dados a partir do teclado também são públicos, podendo ser acessados usando a forma genérica `Keyboard.readXXX`, onde `XXX` é um dos tipos de dados suportados. A listagem B.2 mostra exemplos de uso dos métodos estáticos da classe `Keyboard`.

Listagem B.2: A classe `DemoKeyboard`, que demonstra o uso dos métodos na classe `Keyboard`.

```
1  /**
2   * A classe DemoKeyboard, que demonstra usos dos métodos estáticos na classe Keyboard
3   * para leitura (a partir do teclado ou de arquivos de resposta) de valores de tipos
4   * nativos e de instâncias da classe String.
5   */
6  class DemoKeyboard // declaração da classe
7    {
8    /**
9     * O método main permite a execução desta classe. Este método demonstra usos dos
10    * métodos estáticos na classe Keyboard.
11    * @param argumentos os argumentos que podem ser passados para o método via linha
12    *        de comando, mas que neste caso serão ignorados.
13    */
14   public static void main(String[] argumentos)
15     {
16     // Vamos imprimir mensagens de erros e avisos ou não ?
17     System.out.print("Você quer ver mensagens de erro e avisos ? [s/N]:");
18     // Chama o método com default false, se retornar verdadeiro liga o debug
19     if (Keyboard.readBoolean(false))
20       Keyboard.debugOn();
21     else
22       Keyboard.debugOff();
23     // Leitura de valores booleanos. true, t, yes, y, on, v, s, sim, verdadeiro são
```

```
24      // sinônimos de true. false, f, no, n, off, nao, não, falso são sinônimos de false.
25      boolean b1,b2;
26      System.out.print("Entre um valor booleano ou pressione Enter para [true]:");
27      b1 = Keyboard.readBoolean();
28      System.out.print("Entre um valor booleano ou pressione Enter para [false]:");
29      b2 = Keyboard.readBoolean(false);
30      System.out.println("Você entrou ["+b1+"] e ["+b2+"].");
31      // Leitura de valores do tipo char.
32      char c1,c2;
33      System.out.print("Entre um caracter ou pressione Enter para [' '] (espaço):");
34      c1 = Keyboard.readChar();
35      System.out.print("Entre um caracter ou pressione Enter para ['s']:");
36      c2 = Keyboard.readChar('s');
37      System.out.println("Você entrou ["+c1+"] e ["+c2+"].");
38      // Leitura de valores do tipo byte.
39      byte by1,by2;
40      System.out.print("Entre um valor do tipo byte ou pressione Enter para [0]:");
41      by1 = Keyboard.readByte();
42      System.out.print("Entre um valor do tipo byte ou pressione Enter para [-123]:");
43      by2 = Keyboard.readByte((byte)-123); // notem o cast !
44      System.out.println("Você entrou ["+by1+"] e ["+by2+"].");
45      // Leitura de valores do tipo short.
46      short s1,s2;
47      System.out.print("Entre um valor do tipo short ou pressione Enter para [0]:");
48      s1 = Keyboard.readShort();
49      System.out.print("Entre um valor do tipo short ou pressione Enter para [-9999]:");
50      s2 = Keyboard.readShort((short)-9999); // notem o cast !
51      System.out.println("Você entrou ["+s1+"] e ["+s2+"].");
52      // Leitura de valores do tipo int.
53      int i1,i2;
54      System.out.print("Entre um valor do tipo int ou pressione Enter para [0]:");
55      i1 = Keyboard.readInt();
56      System.out.print("Entre um valor do tipo int ou pressione Enter para [123456]:");
57      i2 = Keyboard.readInt(123456);
58      System.out.println("Você entrou ["+i1+"] e ["+i2+"].");
59      // Leitura de valores do tipo long.
60      long l1,l2;
61      System.out.print("Entre um valor do tipo long ou pressione Enter para [0]:");
62      l1 = Keyboard.readLong();
63      System.out.print("Entre um valor do tipo long ou pressione Enter para [-1000]:");
64      l2 = Keyboard.readLong(-1000); // int como argumento, será convertido para long
65      System.out.println("Você entrou ["+l1+"] e ["+l2+"].");
66      // Leitura de valores do tipo float.
67      float f1,f2;
68      System.out.print("Entre um valor do tipo float ou pressione Enter para [0]:");
69      f1 = Keyboard.readFloat();
70      System.out.print("Entre um valor do tipo float ou pressione Enter para [1e23]:");
71      f2 = Keyboard.readFloat(1e23f);
72      System.out.println("Você entrou ["+f1+"] e ["+f2+"].");
73      // Leitura de valores do tipo double.
74      double d1,d2;
75      System.out.print("Entre um valor do tipo double ou pressione Enter para [0]:");
76      d1 = Keyboard.readDouble();
77      System.out.print("Entre um valor do tipo double ou pressione Enter para [1e-20]:");
78      d2 = Keyboard.readDouble(1e-20);
79      System.out.println("Você entrou ["+d1+"] e ["+d2+"].");
80      // Leitura de instâncias da classe String
81      String st1,st2;
82      System.out.print("Entre uma String ou pressione Enter para [] (vazia):");
83      st1 = Keyboard.readString();
84      System.out.print("Entre uma String ou pressione Enter para [\"Keyboard\"]:");
85      st2 = Keyboard.readString("Keyboard");
86      System.out.println("Você entrou ["+st1+"] e ["+st2+"].");
87      }  // fim do método main
88
89  } // fim da classe DemoKeyboard
```

Os métodos da classe `Keyboard` podem ser usados diretamente em aplicações, como mostrado na listagem B.2, ou em métodos de outras classes, como mostrado na listagem B.3.

Listagem B.3: A classe `DataKeyboard`, cujo construtor lê os valores dos campos do teclado.

```
/**
 * A classe DataKeyboard, que contém um construtor que inicializa a data lendo os
 * seus valores do teclado usando a classe Keyboard. Alguns métodos essenciais e
 * interessantes foram deixados de lado para simplificar - verifique os exercícios
 * do livro para ver como implementá-los. Em particular, esta classe não verifica
 * validade de dias e meses corretamente - ela aceita datas como 31/2/2000.
 */
class DataKeyboard // declaração da classe
  {
 /**
  * Declaração dos campos da classe
  */
  private byte dia,mês; // dia e mês são representados por bytes
  private short ano; // ano é representado por um short

 /**
  * O construtor da classe não recebe argumentos, e lê os valores dos campos usando os
  * métodos da classe Keyboard, verificando até que os valores aceitáveis sejam
  * passados.
  */
  DataKeyboard()
    {
    do // lê o valor do dia até que seja válido
      {
      System.out.print("Entre o dia:");
      dia = Keyboard.readByte();
      }
    while ((dia < 1) || (dia > 31));
    do
      {
      System.out.print("Entre o mês:");
      mês = Keyboard.readByte();
      }
    while ((mês < 1) || (mês > 12));
    System.out.print("Entre o ano:");
    ano = Keyboard.readShort();
    } // fim do construtor

 /**
  * O método mostraData não recebe argumentos nem retorna valores. Este método somente
  * imprime os valores dos campos, formatados de forma que uma barra ("/") seja
  * impressa entre eles. Quando o valor do ano for impresso, uma quebra de linha
  * também será impressa.
  */
  public void mostraData()
    {
    System.out.print(dia);
    System.out.print("/");
    System.out.print(mês);
    System.out.print("/");
    System.out.println(ano);
    } // fim do método mostraData

  } // fim da classe DataKeyboard
```

Uma aplicação que usa a classe `DataKeyboard` é mostrada na listagem B.4.

Listagem B.4: A classe `DemoDataKeyboard`, que demonstra instâncias da classe `DataKeyboard`.

```
/**
 * A classe DemoDataKeyboard, que demonstra usos da classe DataKeyboard.
```

```
3   */
4  class DemoDataKeyboard // declaração da classe
5    {
6    /**
7     * O método main permite a execução desta classe. Este método contém declarações
8     * de algumas instâncias da classe DataKeyboard, e demonstra como usar o construtor
9     * da classe, que lerá seus dados do teclado.
10    * @param argumentos os argumentos que podem ser passados para o método via linha
11    *        de comando, mas que neste caso serão ignorados.
12    */
13   public static void main(String[] argumentos)
14     {
15     // Criamos duas instâncias da classe DataKeyboard, usando a palavra-chave new, que
16     // fará com que o construtor seja chamado.
17     System.out.println("Entre a data de hoje:");
18     DataKeyboard hoje = new DataKeyboard();
19     System.out.println("Entre a data do seu nascimento:");
20     DataKeyboard seuNascimento = new DataKeyboard();
21     // Vamos imprimir os conteúdos das instâncias
22     hoje.mostraData();
23     seuNascimento.mostraData();
24     }  // fim do método main
25
26   } // fim da classe DemoDataKeyboard
```

### B.2.2 Sumário dos métodos da classe `Keyboard`

Uma lista dos métodos públicos da classe `Keyboard`, com informações sobre o uso, é mostrada a seguir:

- O método `readBoolean` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `boolean` e retorna este valor. Este método converte o que for entrado pelo teclado em um valor booleano: se as Strings "`true`", "`t`", "`yes`", "`y`", "`on`", "`v`", "`s`", "`sim`", "`verdadeiro`" (independente de estar em maiúsculas ou minúsculas) forem entradas, o método retornará o valor booleano `true`. Se uma das Strings "`false`", "`f`", "`no`", "`n`", "`off`", "`nao`", "`não`", "`falso`" (independente de estar em maiúsculas ou minúsculas) for entrada, o método retornará o valor booleano `false`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor o método retornará `true`.
  Uma segunda versão deste método aceita um argumento do tipo `boolean` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor.

  *método* `readBoolean`

- O método `readChar` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `char` e retorna este valor. Este método retorna o primeiro caracter do que for entrado pelo teclado, retornando este caracter como um valor do tipo `char`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará um espaço (`' '`).
  Uma segunda versão deste método aceita um argumento do tipo `char` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a leitura não for possível.

  *método* `readChar`

- O método `readByte` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `byte` e retorna este valor. Este método processa o que for entrado pelo teclado, tentando converter este valor para um valor do tipo `byte`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará zero.
  Uma segunda versão deste método aceita um argumento do tipo `byte` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível.

  *método* `readByte`

*método readShort*

- O método `readShort` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `short` e retorna este valor. Este método processa o que for entrado pelo teclado, tentando converter este valor para um valor do tipo `short`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará zero.
  Uma segunda versão deste método aceita um argumento do tipo `short` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível.

*método readInt*

- O método `readInt` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `int` e retorna este valor. Este método processa o que for entrado pelo teclado, tentando converter este valor para um valor do tipo `int`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará zero.
  Uma segunda versão deste método aceita um argumento do tipo `int` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível.

*método readLong*

- O método `readLong` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `long` e retorna este valor. Este método processa o que for entrado pelo teclado, tentando converter este valor para um valor do tipo `long`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará zero.
  Uma segunda versão deste método aceita um argumento do tipo `long` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível.

*método readFloat*

- O método `readFloat` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `float` e retorna este valor. Este método processa o que for entrado pelo teclado, tentando converter este valor para um valor do tipo `float`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará zero.
  Uma segunda versão deste método aceita um argumento do tipo `float` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível.

*método readDouble*

- O método `readDouble` (sem argumentos) aguarda que o usuário entre um valor que pode ser convertido para um valor do tipo `double` e retorna este valor. Este método processa o que for entrado pelo teclado, tentando converter este valor para um valor do tipo `double`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará zero.
  Uma segunda versão deste método aceita um argumento do tipo `double` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível.

*método readString*

- O método `readString` (sem argumentos) aguarda que o usuário entre um valor qualquer pelo teclado e retorna este valor sem processamento adicional, como uma instância da classe `String`. Se o usuário pressione a tecla ENTER sem entrar nenhum valor ou se a conversão não for possível, o método retornará uma String vazia (com comprimento zero).
  Uma segunda versão deste método aceita um argumento que é uma instância da classe `String` que será o valor *default* a ser usado caso o usuário pressione a tecla ENTER sem entrar nenhum valor.

*método debugOn*

- O método `debugOn` (sem argumentos) modifica um campo da classe que faz com que qualquer mensagem de erro ou advertência seja impressa. Este método é util quando é

necessário fazer a verificação de erros em entrada de dados, mas deve ser raramente usado para aplicações comuns.

- O método `debugOff` (sem argumentos) modifica um campo da classe que faz com que qualquer mensagem de erro ou advertência não seja impressa, o que é o comportamento normal da classe.

*método* `debugOff`

## B.3 Usando arquivos de respostas

A possibilidade de entrar dados via teclado aumenta a flexibilidade de programas sem interface gráfica ou outro mecanismo de entrada de dados. Em contrapartida, a execução destes programas pode ser tediosa caso muitos dados devam ser entrados.

A classe `Keyboard` oferece a possibilidade de usarmos *arquivos de respostas*, que são arquivos simples de texto contendo, em cada linha, um dado cuja digitação será simulada através do redirecionamento de arquivos.

*arquivos de respostas*

Para exemplificar, consideremos a classe `DemoDataKeyboard`, que espera que o usuário digite seis dados: um dia, mês e ano para a primeira data e outro dia, mês e ano para a segunda data. O arquivo mostrado na listagem B.5 poderia substituir o processo de digitação.

Listagem B.5: Um exemplo de arquivo de respostas para uso com a classe `DemoDataKeyboard`.

```
14
07
2001
28
03
1966
```

Para que este arquivo seja usado ao invés da digitação direta dos dados devemos *redirecionar* este arquivo para servir de entrada à classe `DemoDataKeyboard`. Isto é feito quando usamos a máquina virtual para executar a classe, usando o sinal "`<`" para fazer o redirecionamento. Se por exemplo, o arquivo de respostas se chamar `datas.dat` podemos fazer o redirecionamento com o comando `java DemoDataKeyboard < datas.dat`. A sintaxe do comando é a mesma para os sistemas operacionais Windows (usando a linha de comando) e Linux.

*redirecionamento de arquivos*

O suporte a arquivos de respostas é limitado, seu uso somente funcionará sem problemas se todos os dados a serem digitados forem entrados no arquivo de respostas, um em cada linha. Exceções ocorrerão caso estas regras não sejam seguidas.

# Referências Bibliográficas

[1] transformar em bibliografia comentada, talvez seção à parte

[2] Ken Arnold, James Goslin, *Programando em Java*, Makron Books, 1997.

[3] Samuel N. Kamin, M. Dennis Mickunas, Edward M. Reingold, *An Introduction to Computer Science Using Java*, WCB/McGraw-Hill, 1998.

[4] David Arnow, Gerald Weiss, *Introduction to Programming Using Java - An Object-Oriented Approach*, Addison-Wesley, 1998.

[5] Patrick Chan, Rosanna Lee, Douglas Kramer, *The Java Class Libraries, Second Edition, Volume One*, Addison-Wesley, 1998.

[6] Patrick Chan, Rosanna Lee, *The Java Class Libraries, Second Edition, Volume Two*, Addison-Wesley, 1998.

[7] Patrick Chan, Rosanna Lee, Douglas Kramer, *The Java Class Libraries, Second Edition, Volume One, Supplement for the Java 2 Platform, Standard Edition, v1.2*, Addison-Wesley, 1999.

[8] David M. Geary, *Graphic Java - Mastering the AWT*, Sun Microsystems Press/Prentice Hall, 1997.

[9] John Withers, *Developing Java Entertainment Applets*, John Wiley and Sons, 1997.

[10] Jean-Paul Tremblay, Richard B. Bunt, *Introduction to Computer Science - An Algorithmic Approach*, McGraw-Hill, 1979.

[11] Jean-Paul Tremblay, Richard B. Bunt, *Instructor's manual to accompany An Introduction to Computer Science - An Algorithmic Approach*, McGraw-Hill, 1979.

[12] Simon Roberts, Philip Heller, Michael Ernest, *Complete Java 2 Certification Study Guide*, Sybex, 2000.

[13] Lawrence H. Rodrigues, *Building Imaging Applications with Java Technology*, Addison-Wesley, 2001.

[14] Michael T. Goodrich, Roberto Tamassia, *Data Structures and Algorithms in Java*, John Wiley and Sons, Inc., 1998.

[15] Cay S. Horstmann, Gary Cornell, *Core Java 2 - Fundamentals*, Prentice Hall, 2001.

[16] Cay S. Horstmann, Gary Cornell, *Core Java 2 - Fundamentals*, Prentice Hall, 2001.

[17] David Harel, *Algoruthmics - The Spirit of Computing, second edition*, Addison-Wesley, 1992.

[18] Andreas D. Baxevanis, B. F. Francis Ouellette, *Bioinformatics - A Practical Guide to the Analysis of Genes and Proteins*, Wiley-Liss, 1998.

[19] João Carlos Setúbal, João Meidanis, *Introduction to Computational Molecular Biology*, PWS Publishing Company, 1997.

[20] Dan Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambrigde University Press, 1997.

# Índice Remissivo





Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.

Versão 182 compilada em 30 de julho de 2001. Visite o site de apoio em http://www.bioinfo.univap.br/~POO.