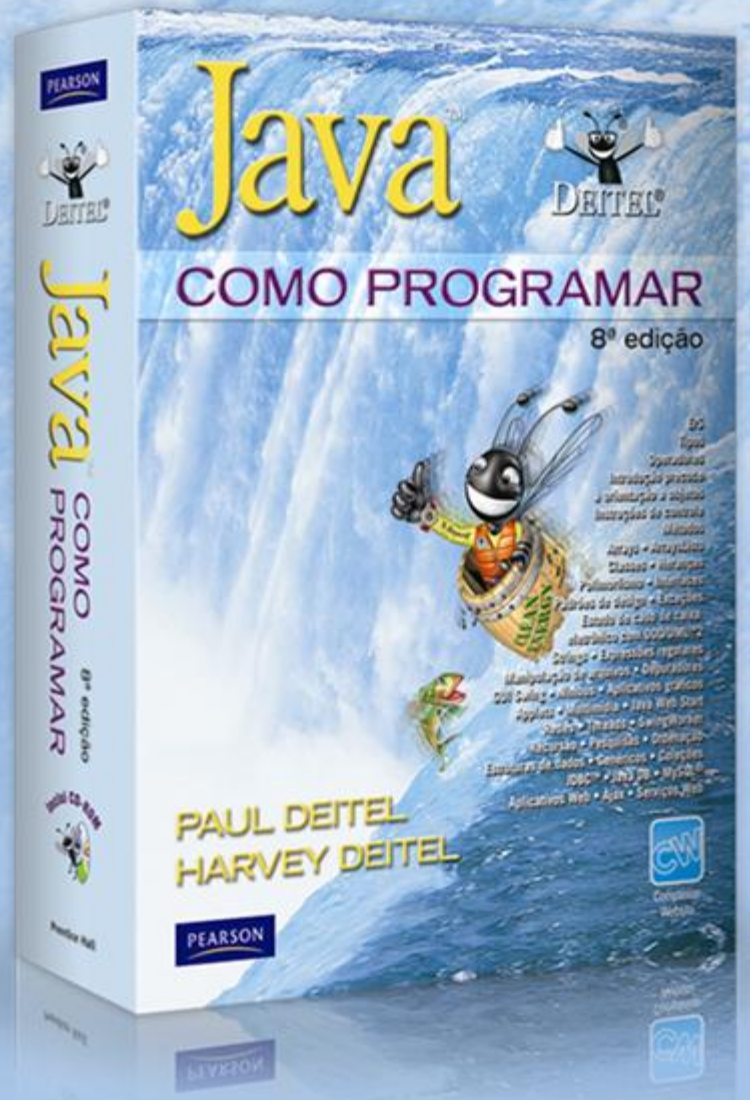
Capítulo 6
Métodos: uma visão mais aprofundada



Java™ Como Programar, 8/E

OBJETIVOS

Neste capítulo, você aprenderá:

- Como métodos e campos `static` são associados a uma classe inteira em vez de instâncias específicas da classe.
- A utilizar métodos `Math` comuns disponíveis na Java API.
- A entender os mecanismos para passar informações entre métodos.
- Como o mecanismo de chamada/retorno de método é suportado pela pilha de chamadas de métodos e registros de ativação.
- Como pacotes agrupam classes relacionadas.
- Como utilizar a geração de números aleatórios para implementar aplicativos de jogos de azar.
- Como a visibilidade das declarações é limitada a regiões específicas dos programas.
- O que é a sobrecarga de método e como criar métodos sobrecarregados.

6.1 Introdução

- A melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas e simples partes, ou **módulos**.
  - **dividir para conquistar.**
- Tópicos neste capítulo:
  - Métodos `static`
  - Declare um método com mais de um parâmetro.
  - Pilha de chamadas de método.
  - Técnicas de simulação com geração de números aleatórios.
  - Como declarar valores que não podem mudar (isto é, constantes) nos seus programas.
  - Sobrecarga de método.

6.2 Módulos de programa em Java

- Programas Java combinam novos métodos e classes que o você escreve com métodos e classes predefinidos disponíveis na **Java Application Programming Interface** e em outras bibliotecas de classes.
- Em geral, classes relacionadas são agrupadas em pacotes de modo que possam ser importadas nos programas e reutilizadas.
  - Você aprenderá a agrupar suas próprias classes em pacotes no Capítulo 8.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.1

Familiarize-se com a rica coleção de classes e métodos fornecidos pela Java API (`java.sun.com/javase/6/docs/api/`*). Na Seção 6.8, apresentamos uma visão geral dos vários pacotes comuns. No Apêndice E, explicamos como navegar pela documentação da Java API. Não reinvente a roda. Quando possível, reutilize as classes e métodos na Java API. Isso reduz o tempo de desenvolvimento de programas e evita a introdução de erros.*

- Métodos ajudam a modularizar um programa separando suas tarefas em unidades autocontidas.
- Instruções no corpo dos métodos:
  - Escrito somente uma vez.
  - Ocultado de outros métodos.
  - Pode ser reutilizado a partir de de várias localizações em um programa.
- Abordagem "dividir para conquistar".
  - Construindo programas a partir de pequenos e simples fragmentos
- **Capacidade de reutilização de software**
  - Use métodos existentes como blocos de construção para criar novos programas.
- Dividir um programa em métodos significativos torna o programa mais fácil de depurar e manter.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.2

Para promover a capacidade de reutilização de software, todos os métodos devem estar limitados à realização de uma única tarefa bem-definida e o nome do método deve expressar essa tarefa efetivamente.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 6.1

Um método que realiza uma única tarefa é mais fácil de testar e depurar do que aquele que realiza muitas tarefas.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.3

Se você não puder escolher um nome conciso que expresse a tarefa de um método, seu método talvez tente realizar tarefas em demasia. Divida esse método em vários métodos menores.

- Forma hierárquica de gerenciamento. (Figura 6.1).
  - Um chefe (o chamador) solicita que um trabalhador (o método chamado) realize uma tarefa e informe (retorne) os resultados depois de completar a tarefa.
  - O método chefe não tem conhecimento sobre como o método trabalhador realiza suas tarefas designadas.
  - O trabalhador também pode chamar outros métodos trabalhadores, sem o que chefe saiba.
- Esse "ocultamento" dos detalhes da implementação promove a boa engenharia de software.

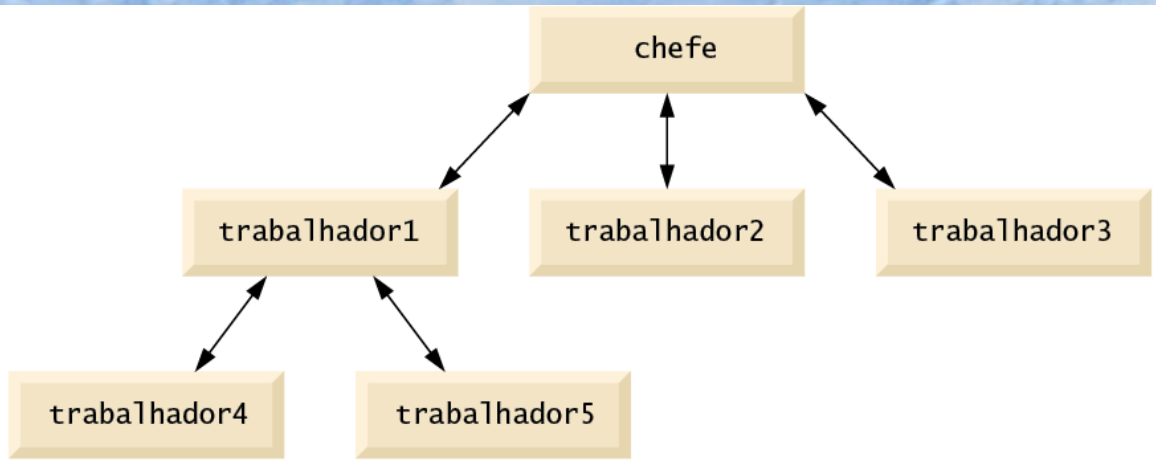


Figura 6.1 | Relacionamento hierárquico entre método trabalhador e método chefe.

6.3 Métodos `static`, campos `static` e a classe `Math`

- Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto.
  - Aplica-se à classe na qual é declarado como um todo.
  - Conhecido como método `static` ou **método de classe.**
- É comum uma classe conter um grupo de métodos `static` convenientes para realizar tarefas comuns.
- Para declarar um método como `static`, coloque a palavra-chave `static` antes do tipo de retorno na declaração do método.
- Chamando um método `static`
  - *NomeDaClasse .nomeDoMétodo (argumentos)*
- A classe `Math` fornece uma coleção de métodos `static` que permitem realizar cálculos matemáticos comuns.
- Os argumentos de método podem ser constantes, variáveis ou expressões.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.4

A classe `Math` *faz parte do pacote* `java.lang`*, que é implicitamente importado pelo compilador, assim não é necessário importar a classe* `Math` *para utilizar seus métodos.*

Método	Descrição	Exemplo

`abs( x )`	valor absoluto de *x*	`abs( 23.7 )` é `23.7`
		`abs( 0.0 )` é `0.0`
		`abs( -23.7 )` é `23.7`

`ceil( x )`	arredonda *x* para o menor inteiro não menor que *x*	`ceil( 9.2 )` é `10.0` `ceil( -9.8 )` é `-9.0`
`cos( x )`	co-seno trigonométrico de *x* (*x* em radianos)	`cos( 0.0 )` é `1.0`
`exp( x )`	método exponencial e^x	`exp( 1.0 )` é `2.71828` `exp( 2.0 )` é `7.38906`
`floor( x )`	arredonda *x* para o maior inteiro não maior que *x*	`floor( 9.2 )` é `9.0` `floor( -9.8 )` é `-10.0`
`log( x )`	logaritmo natural de *x* (base *e*)	`log( Math.E )` é `1.0` `log( Math.E* Math.E )` é `2.0`
`max(x,y)`	maior valor de *x* e *y*	`max( 2.3, 12.7 )` é `12.7` `max( -2.3, -12.7 )` é `-2.3`
`min(x,y)`	menor valor de *x* e *y*	`min( 2.3, 12.7 )` é `2.3` `min( -2.3, -12.7 )` é `-12.7`

Figura 6.2 | Métodos da classe `Math`. (Parte 1 de 2.)

Método	Descrição	Exemplo
`pow(x,y)`	*x* elevado à potência de *y* (isto é, x^y)	`pow( 2.0, 7.0 )` é `128.0` `pow( 9.0, 0.5 )` é `3.0`
`sin( x )`	seno trigonométrico de *x* (*x* em radianos)	`sin( 0.0 )` é `0.0`
`sqrt( x )`	raiz quadrada de *x*	`sqrt( 900.0 )` é `30.0`
`tan( x )`	tangente trigonométrica de *x* (*x* em radianos)	`tan( 0.0 )` é `0.0`

Figura 6.2 | Métodos da classe `Math`. (Parte 2 de 2.)

- Campos `Math` para constantes matemáticas comuns
  - **Math.PI** (3.141592653589793)
  - **Math.PI** (2.718281828459045)
- Declarada na classe `Math` como os modificadores `public`, `final` e `static`
  - `public` permite usar esses campos em suas próprias classes.
  - Qualquer campo declarado com a palavra-chave **final** é constante — seu valor não pode ser alterado depois que o campo é inicializado.
  - `PI` e `E` são declarados `final` porque seus valores nunca mudam.

- O campo que representa o atributo também é conhecido como uma variável de instância — cada objeto de uma classe mantém sua própria cópia de um atributo na memória.
- Campos em que cada objeto de uma classe não tem uma instância separada do campo são declarados `static` e também são conhecidos como **variáveis de classe**.
- Todos os objetos de uma classe que contêm campos `static` compartilham uma cópia desses campos.
- As variáveis de classe e as variáveis de instância (isto é, variáveis `static`) representam os campos de uma classe.

- Por que o método `main` é declarado `static`?
  - A JVM tenta invocar o método `main` da classe que você especifica — quando nenhum objeto da classe foi criados.
  - Declarar `main` como `static` permite que a JVM invoque `main` sem criar uma instância da classe.

6.4 Declarando métodos com múltiplos parâmetros

- Múltiplos parâmetros são especificados como uma lista separada por vírgulas.
- Deve haver um argumento na chamada de método para cada parâmetro (às vezes chamado **parâmetro formal**) na declaração de método.
- Cada argumento deve ser consistente com o tipo do parâmetro correspondente.

```
1  // Figura 6.3: MaximumFinder.java
2  // Método maximum declarado pelo programador com três parâmetros double.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtém três valores de ponto flutuante e localiza o valor máximo
8     public void determineMaximum()
9     {
10       // cria Scanner para entrada a partir da janela de comando
11       Scanner input = new Scanner( System.in );
12
13       // solicita e insere três valores de ponto flutuante
14       System.out.print(
15          "Enter three floating-point values separated by spaces: " );
16       double number1 = input.nextDouble(); // lê o primeiro double
17       double number2 = input.nextDouble(); // lê o segundo double
18       double number3 = input.nextDouble(); // lê o terceiro double
19
20       // determina o valor máximo
21       double result = maximum( number1, number2, number3 );
22
```

Passa três argumentos para o método `maximum`

Figura 6.3 | O método `maximum` declarado pelo programador com três parâmetros `double`. (Parte 1 de 2.)

```
23          // exibe o valor máximo
24          System.out.println( "Maximum is: " + result );
25       } // fim do método determineMaximum
26
27       // retorna o máximo dos seus três parâmetros de double
28       public double maximum( double x, double y, double z )
29       {
30          double maximumValue = x; // supõe que x é o maior valor inicial
31
32          // determina se y é maior que maximumValue
33          if ( y > maximumValue )
34             maximumValue = y;
35
36          // determina se z é maior que maximumValue
37          if ( z > maximumValue )
38             maximumValue = z;
39
40          return maximumValue;
41       } // fim do método Maximum
42    } // fim da classe MaximumFinder
```

O método `maximum` recebe três parâmetros e retorna o maior dos três

Figura 6.3 | O método `maximum` declarado pelo programador com três parâmetros `double`. (Parte 2 de 2.)


```
1  // Figura 6.4: MaximumFinderTest.java
2  // Aplicativo para testar a classe MaximumFinder.
3
4  public class MaximumFinderTest
5  {
6     // ponto de partida do aplicativo
7     public static void main( String[] args )
8     {
9        MaximumFinder maximumFinder = new MaximumFinder();
10       maximumFinder.determineMaximum();
11    } // fim de main
12 } // fim da classe MaximumFinderTest
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Figura 6.4 | Aplicativo para testar a classe `MaximumFinder`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.5

Métodos podem retornar no máximo um valor, mas o valor retornado poderia ser uma referência a um objeto que contém muitos valores.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.6

Variáveis devem ser declaradas como campos de uma classe somente se forem utilizadas em mais de um método da classe ou se o programa deve salvar seus valores entre chamadas aos métodos da classe.

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.1

Declarar parâmetros de método do mesmo tipo como `float x, y` *em vez de* `float x, float y` *é um erro de sintaxe — um tipo é requerido para cada parâmetro na lista de parâmetros.*

- Implementando o método `maximum` reutilizando o método `Math.max`
  - Duas chamadas a `Math.max`, da seguinte maneira:
    - `return Math.max( x, Math.max( y, z ) );`
  - O primeiro especifica os argumentos `x` e `Math.max(y,z)`.
  - Antes de qualquer chamada de método, seus argumentos devem ser avaliados para determinar seus valores.
  - Se um argumento for uma chamada de método, a chamada de método deve ser realizada para determinar seu valor de retorno.
  - O resultado da primeira chamada é passado como o segundo argumento para a outra chamada, o que retorna o maior dos seus dois argumentos.

- **Concatenação de string**
  - Monte objetos `String` em strings maiores com os operadores + ou +=.
- Quando ambos os operandos do operador + são `String`, o operador + cria um novo objeto `String`.
  - os caracteres do operando direito são colocados no fim daqueles no operando à esquerda.
- Todos os objetos e valores primitivos em Java têm uma representação `String`.
- Quando um dos operandos do operador + for uma `String`, o outro é convertido em uma `String` e então os dois são concatenados.
- Se um `boolean` é concatenado com uma `String`, o `boolean` é convertido em uma `String` `"true"` ou `"false"`.
- Todos os objetos têm um método `toString` que retorna uma representação `String` do objeto.

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.2

É um erro de sintaxe dividir um literal de `String` *em linhas. Se necessário, você pode dividir uma* `String` *em várias unidades menores e utilizar concatenação para formar a* `String` *desejada.*

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.3

Confundir o operador + utilizado para concatenação de string com o operador + utilizado para adição pode levar a resultados estranhos. O Java avalia os operandos de um operador da esquerda para a direita. Por exemplo, suponha que a variável inteira y tem o valor 5, a expressão "y + 2 = " + y + 2 resulta na string "y + 2 = 52", não em "y + 2 = 7", porque o primeiro valor de y (5) é concatenado na string "y + 2 = ", então o valor 2 é concatenado na nova e maior string "y + 2 = 5". A expressão "y + 2 = " + (y + 2) produz o resultado desejado "y + 2 = 7".

6.5 Notas sobre a declaração e utilização de métodos

- Três maneiras de chamar um método:
  - Utilizando o nome de um método sozinho para chamar outro método da mesma classe.
  - Utilizando uma variável que contém uma referência a um objeto, seguido por um ponto (`.`) e o nome do método para chamar um método do objeto referenciado.
  - Utilizando o nome da classe e um ponto (`.`) para chamar um método `static` de uma classe.

- Um método não `static` pode chamar qualquer método da mesma classe diretamente e pode manipular qualquer um dos campos da classe diretamente.
- Um método `static` pode chamar *somente outros métodos* `static` da mesma classe diretamente e pode manipular *somente campos* `static` na mesma classe diretamente.
  - Para acessar os membros não `static` da classe, um método `static` deve utilizar uma referência a um objeto da classe.

- Três maneiras de retornar o controle à instrução que chama um método:
  - Quando o fluxo de programa alcança a chave direita que termina o método.
  - Quando a seguinte instrução executa

    ```
    return;
    ```

  - Quando o método devolve um resultado com uma instrução como `return` *expressão*`;`

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.4

A classe `Math` *faz parte do pacote* `java.lang`*, que é implicitamente importado pelo compilador, assim não é necessário importar a classe* `Math` *para utilizar seus métodos.*

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.5

Omitir o tipo do valor de retorno, possivelmente `void`*, em uma declaração de método é um erro de sintaxe.*

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.6

Colocar um ponto-e-vírgula após o parêntese direito que envolve a lista de parâmetros de uma declaração de método é um erro de sintaxe.

Java™ COMO PROGRAMAR 8ª edição

DEITEL®

Erro de programação comum 6.7

Redeclarar um parâmetro como uma variável local no corpo do método é um erro de compilação.

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.8

Esquecer de retornar um valor em um método que deve retornar um valor é um erro de compilação. Se um tipo de retorno além de `void` *for especificado, o método deverá conter uma instrução* `return` *que retorne um valor consistente com o tipo de retorno do método. Retornar um valor de um método cujo tipo de retorno foi declarado como* `void` *é um erro de compilação.*

6.6 Pilha de chamadas de método e registros de ativação

- Estrutura de dados **pilha.**
  - Análoga a uma pilha de pratos
  - Quando um prato é colocado na pilha, ele é sempre colocado na parte superior (processo referido como **adicionar** o prato na pilha).
  - Quando um prato é colocado na pilha, ele é sempre colocado na parte superior (processo referido como **inserir** o prato na pilha).
- **Estruturas de dados "último a entrar, primeiro a sair" (*Last-In, First-Out* – LIFO)**
  - O último item inserido na pilha é o primeiro item removido da pilha.

- Quando um programa chama um método, o método chamado deve saber como retornar ao seu chamador.
  - O endereço de retorno do método chamador é inserido na **pilha de execução de programas** (ou **pilha de chamadas de método**).
- Se uma série de chamadas de método ocorrer, os sucessivos endereços de retorno são inseridos na pilha na ordem "último a entrar, primeiro a sair".
- A pilha de execução de programas também contém a memória para as variáveis locais utilizadas em cada invocação de um método durante uma execução do programa.
  - Armazenados como uma parte da pilha de execução de programas, é conhecida como **registro de ativação** ou **stack frame ("quadro de pilha")** das chamadas de método.

- Quando uma chamada de método é feita, o registro de ativação dessa chamada de método é inserido na pilha de execução de programas.
- Quando o método retorna ao seu chamador, o registro de ativação do método é retirado da pilha e essas variáveis locais não são mais conhecidas para o programa.
- Se mais chamadas de método ocorrerem além do limite de seus registros de ativação armazenados na pilha de execução de programas, ocorrerá um erro conhecido como **estouro de pilha**.

6.7 Promoção e coerção de argumentos

- **Promoção de argumentos.**
  - Converter um valor de um argumento para o tipo que o método espera receber no seu parâmetro correspondente.
- Conversões podem levar a erros de compilação se as **regras de promoção** do Java não forem cumpridas.
- Regras de promoção.
  - especifica as conversões que são permitidas.
  - aplicam-se a expressões que contém valores de dois ou mais tipos primitivos e a valores de tipo primitivo passados como argumentos para os métodos.
- Cada valor é promovido para o tipo "mais alto" na expressão.
- A Figura 6.5 lista os tipos primitivos e os tipos para os quais cada um pode ser promovido.

Tipo	Promoções válidas
`double`	None
`float`	`double`
`long`	`float` ou `double`
`int`	`long`, `float` ou `double`
`char`	`int`, `long`, `float` ou `double`
`short`	`int`, `long`, `float` ou `double` (mas não `char`)
`byte`	`short`, `int`, `long`, `float` ou `double` (mas não `char`)
`boolean`	Nenhuma (os valores `boolean` não são considerados como números em Java)

Figura 6.5 | Promoções permitidas para tipos primitivos.

- Converter valores em tipos mais baixos na tabela da Figura 6.5 resultará em diferentes valores se o tipo mais baixo não puder representar o valor do tipo mais alto.
- Nos casos em que as informações podem ser perdidas devido à conversão, o compilador Java requer que você utilize um operador de coerção explicitamente para forçar a conversão a ocorrer — do contrário, ocorre um erro de compilação.

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.9

Converter um valor de tipo primitivo em um outro tipo primitivo pode alterar o valor se o novo tipo não for uma promoção válida. Por exemplo, converter um valor de ponto flutuante em um valor inteiro pode introduzir erros de truncamento (perda da parte fracionária) no resultado.

6.8 Pacotes de Java API

- O Java contém muitas classes predefinidas que são agrupadas em categorias de classes relacionadas chamadas pacotes.
- Uma grande vantagem do Java são as milhares de classes da Java API.
- Alguns pacotes chave de Java API são descritos na Figura 6.6.
- Visão geral dos pacotes no Java SE 6:
  - `java.sun.com/javase/6/docs/api/overview-summary.html`
- Documentação da Java API
  - `java.sun.com/javase/6/docs/api/`

Pacote	Descrição
`java.applet`	O **Java Applet Package** contém uma classe e várias interfaces exigidas para criar applets Java — programas que executam nos navegadores da Web. Os applets são discutidos no Capítulo 23, "Applets e Java Web Start"; as interfaces são discutidas no Capítulo 10, "Programação orientada a objetos: polimorfismo".
`java.awt`	O **Java Abstract Window Toolkit Package** contém as classes e interfaces exigidas para criar e manipular GUIs em versões anteriores do Java. Nas versões atuais do Java, os componentes GUI Swing dos pacotes `javax.swing` são geralmente usados em seu lugar. (Alguns elementos do pacote `java.awt` são discutidos no Capítulo 14, "Componentes GUI: Parte 1", Capítulo 15, "Imagens gráficas e Java2D™" e Capítulo 25, "Componentes GUI: Parte 2".)
`java.awt.event`	O **Java Abstract Window Toolkit Event Package** contém classes e interfaces que permitem o tratamento de eventos para componentes GUI tanto nos pacotes `java.awt` como `javax.swing`. (Ver o Capítulo 14, "Componentes GUI: Parte 1" e o Capítulo 25, "Componentes GUI: Parte 2".)

Figura 6.6 | Pacotes da Java API (um subconjunto). (Parte 1 de 4.)

Pacote	Descrição
`java.awt.geom`	O **Java 2D Shapes Package** contém classes e interfaces para trabalhar com as avançadas capacidades gráficas bidimensionais do Java. (Consulte o Capítulo 15, "Imagens gráficas e Java 2D™".)
`java.io`	O **Java Input/Output Package** contém classes e interfaces que permitem aos programas gerar entrada e saída de dados. (Consulte o Capítulo 17, "Arquivos, fluxos e serialização de objetos".)
`java.lang`	O **Java Language Package** contém classes e interfaces (discutidas em todo o livro) que são exigidas por muitos programas Java. Esse pacote é importado pelo compilador em todos os programas.
`java.net`	O **Java Networking Package** contém classes e interfaces que permitem aos programas comunicar-se via redes de computadores, como a Internet. (Consulte o Capítulo 27, "Redes".)
`java.sql`	O **JDBC Package** contém classes e interfaces para trabalhar com bancos de dados. (Consulte o Capítulo 28, "Acesso a bancos de dados com o JDBC".)

Figura 6.6 | Pacotes da Java API (um subconjunto). (Parte 2 de 4.)

Pacote	Descrição
`java.text`	O **Java Text Package** contém classes e interfaces que permitem aos programas manipular números, datas, caracteres e strings. O pacote fornece recursos de internacionalização que permitem a um programa ser personalizado para localidades (por exemplo, um programa pode exibir strings em diferentes idiomas com base no país do usuário).
`java.util`	O **Java Utilities Package** contém classes utilitárias e interfaces que permitem ações como manipulações de data e hora, processamento de número aleatório (classe `Random`) e o armazenamento e o processamento de grandes quantidades de dados. (Consulte o Capítulo 20, "Coleções genéricas".)
`java.util.concurrent`	O **Java Concurrency Package** contém classes utilitárias e interfaces para implementar programas que podem realizar múltiplas tarefas paralelamente. (Consulte o Capítulo 26, "Multithreading".)
`javax.media`	O **Java Media Framework Package** contém classes e interfaces para trabalhar com capacidades multimídia do Java. (Consulte o Capítulo 24, "Multimídia: Applets e aplicativos".)

Figura 6.6 | Pacotes da Java API (um subconjunto). (Parte 3 de 4.)

Pacote	Descrição
`javax.swing`	O **Java Swing GUI Components Package** contém classes e interfaces para componentes GUI Swing do Java que fornecem suporte para GUIs portáveis. (Ver o Capítulo 14, "Componentes GUI: Parte 1" e no Capítulo 25: "Componentes GUI: Parte 2".)
`javax.swing.event`	O **Java Swing Event Package** contém classes e interfaces que permitem o tratamento de eventos (por exemplo, responder a cliques de botão) para componentes GUI do pacote `javax.swing`. (Ver o Capítulo 14, "Componentes GUI: Parte 1" e no Capítulo 25, "Componentes GUI: Parte 2".)
`javax.xml.ws`	O **JAX-WS Package** contém classes e interfaces para trabalhar com serviços da Web no Java. (Consulte o Capítulo 31, "Serviços da Web".)

Figura 6.6 | Pacotes da Java API (um subconjunto). (Parte 4 de 4.)

- Simulação e execução de jogos.
  - **Elemento chance**
  - Classe **Random** (pacote `java.util`)
  - Método `static random` da classe `Math`.
- Objetos da classe `Random` podem produzir valores `boolean`, `byte`, `float`, `double`, `int`, `long` e Gaussian aleatórios
- O método `Math random` pode produzir apenas valores `double` no intervalo `0.0` $\leq$ x $<$ `1.0`.
- Documentação para a classe `Random`
  - `java.sun.com/javase/6/docs/api/java/util/Random.html`

- A classe `Random` produz **números pseudoaleatórios**
  - Uma sequência de valores produzida por um cálculo matemático complexo.
  - O cálculo usa a hora atual para **semear** o gerador de números aleatórios.
- O intervalo de valores produzido diretamente pelo método `Random` `nextInt` costuma diferir do intervalo de valores requerido em um aplicativo Java particular.
- O método `Random` `nextInt` que recebe um argumento `int` e retorna um valor entre 0 e, sem incluir, o valor do argumento.

- Lançando um dado de seis faces
  - `face = 1 + randomNumbers.nextInt( 6 );`
  - O argumento `6` — chamado **fator de escalonamento** — representa o número de valores únicos que `nextInt` deve produzir (0–5)
  - Isso se chama **escalonar** o intervalo de valores.
  - Um dado de seis lados tem os números 1–6 nas suas faces, não 0–5.
  - Nós **deslocamos** o intervalo dos números produzidos adicionando um **valor de deslocamento** — nesse caso 1 — para nosso resultado anterior, como em
  - `face = 1 + randomNumbers.nextInt( 6 );`
  - O valor de deslocamento (`1`) especifica o primeiro valor no intervalo desejado de inteiros aleatórios.


```
1  // Figura 6.7: RandomIntegers.java
2  // Inteiros aleatórios deslocados e escalonados.
3  import java.util.Random; // o programa utiliza a classe Random
4
5  public class RandomIntegers
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // gerador de número aleatório
10       int face; // armazena cada inteiro aleatório gerado
11
12       // faz o loop 20 vezes
13       for ( int counter = 1; counter <= 20; counter++ )
14       {
15          // seleciona o inteiro aleatório entre 1 a 6
16          face = 1 + randomNumbers.nextInt( 6 );
17
18          System.out.printf( "%d  ", face ); // exibe o valor gerado
19
20          // se o contador for divisível por 5, inicia uma nova linha de saída
21          if ( counter % 5 == 0 )
22             System.out.println();
23       } // for final
24    } // fim de main
25 } // fim da classe RandomIntegers
```

O programa usa a classe Random do pacote `java.util`

Cria o objeto Random

Produz inteiros no intervalo 1 a 6

Figura 6.7 | Inteiros aleatórios deslocados e escalonados. (Parte 1 de 2.)

```
1  5  3  6  2
5  2  6  5  2
4  4  4  2  6
3  1  6  2  2
```

```
6  5  4  2  6
1  2  5  1  3
6  3  2  2  1
6  4  2  6  4
```

Figura 6.7 | Inteiros aleatórios deslocados e escalonados. (Parte 2 de 2.)

```
Face    Frequency
1       982
2       1001
3       1015
4       1005
5       1009
6       988
```

```
Face    Frequency
1       1029
2       994
3       1017
4       1007
5       972
6       981
```

Figura 6.8 | Rolando um dado de seis lados 6.000 vezes.


```
1  // Figura 6.8: RollDie.java
2  // Rola um dado de seis lados 6 mil vezes.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String[] args )
8     {
9        Random randomNumbers = new Random(); // gerador de número
10                                            // aleatório
11       int frequency1 = 0; // mantém a contagem de 1s lançados
12       int frequency2 = 0; // contagem de 2s lançados
13       int frequency3 = 0; // contagem de 3s lançados
14       int frequency4 = 0; // contagem de 4s lançados
15       int frequency5 = 0; // contagem de 5s lançados
16       int frequency6 = 0; // contagem de 6s lançados
17
18       int face; // armazena o valor lançado mais recentemente
19
20       // soma 6.000 lançamentos de um dado
21       for ( int roll = 1; roll <= 6000; roll++ )
22       {
23          face = 1 + randomNumbers.nextInt( 6 ); // número entre 1 a 6
24
```

Figura 6.8 | Rolando um dado de seis lados 6.000 vezes. (Parte 1 de 3.)

```
25          // define o valor de lançamento de 1 a 6 e incrementa o contador apropriado
26          switch ( face )
27          {
28             case 1:
29                ++frequency1; // incrementa o contador de 1s
30                break;
31             case 2:
32                ++frequency2; // incrementa o contador de 2s
33                break;
34             case 3:
35                ++frequency3; // incrementa o contador de 3s
36                break;
37             case 4:
38                ++frequency4; // incrementa o contador de 4s
39                break;
40             case 5:
41                ++frequency5; // incrementa o contador de 5s
42                break;
43             case 6:
44                ++frequency6; // incrementa o contador de 6s
45                break; // opcional no final do switch
46          } // fim do switch
47       } // for final
48
```

Valor de de 1 a 6 que incrementa o contador apropriado

Figura 6.8 | Rolando um dado de seis lados 6.000 vezes. (Parte 2 de 3.)


```
49          System.out.println( "Face\tFrequency" ); // cabeçalhos de saída
50          System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51             frequency1, frequency2, frequency3, frequency4,
52             frequency5, frequency6 );
53       } // fim de main
54    } // fim da classe RollDie
```

```
Face    Frequency
1       982
2       1001
3       1015
4       1005
5       1009
6       988
```

```
Face    Frequency
1       1029
2       994
3       1017
4       1007
5       972
6       981
```

Figura 6.8 | Rolando um dado de seis lados 6.000 vezes. (Parte 3 de 3.)

6.9.1 Escalonamento e deslocamento generalizados de números aleatórios

- Generaliza o ajuste de escala e deslocamento de números aleatórios:

```
number = valorDeDeslocamento +
         randomNumbers.nextInt( fatorDeEscalonamento );
```

onde *valorDeDeslocamento* especifica o primeiro número no intervalo desejado de inteiros consecutivos e *fatorDeEscalonamento* especifica quantos números estão no intervalo.

- Também é possível escolher inteiros aleatoriamente a partir de conjuntos de valores além dos intervalos de inteiros consecutivos:

```
number = valorDeDeslocamento + diferençaEntreOsValores *
         randomNumbers.nextInt( fatorDeEscalonamento );
```

onde *valorDeDeslocamento* especifica o primeiro número no intervalo desejado de valores, *diferençaEntreValores* representa a diferença entre números consecutivos na sequência e *fatorDeEscalonamento* especifica quantos números estão no intervalo.

6.9.2 Repetição de números aleatórios para teste e depuração

- Ao depurar um aplicativo, às vezes é útil repetir a mesma exata sequência de números pseudoaleatórios.
- Para fazer isso, crie um objeto `Random` da seguinte maneira:
  - ```
    Random randomNumbers =
        new Random( valorSemeado );
    ```
    - O argumento `valorSemeado` (tipo `long`) semeia o cálculo de números aleatórios.
- Você pode configurar a semente de um objeto `Random` em qualquer momento durante a execução do programa chamando o método `set` do objeto.
    ```

Dica de prevenção de erro 6.2

Ao desenvolver um programa, crie o objeto Random *com um valor específico de semente para produzir uma sequência repetível de números toda vez que o programa for executado. Se ocorrer um erro de lógica, corrija-o e teste o programa novamente com o mesmo valor de semente — isso permite reconstruir a mesma sequência de números que causou o erro. Depois que os erros de lógica foram removidos, crie o objeto* Random *sem utilizar um valor de semente, fazendo com que o objeto* Random *gere uma nova sequência de números aleatórios toda vez que o programa é executado.*

6.10 Estudo de caso: Um jogo de azar; introdução a enumerações

- Regras básicas do jogo de dados Craps:
  - *Você lança dois dados. Cada dado tem seis faces que contêm um, dois, três, quatro, cinco e seis pontos, respectivamente. Depois que os dados param de rolar, a soma dos pontos nas faces viradas para cima é calculada. Se a soma for 7 ou 11 no primeiro lance, você ganha. Se a soma for 2, 3 ou 12 no primeiro lance (chamado "craps"), você perde (isto é, a "casa" ganha) . Se a soma for 4, 5, 6, 8, 9 ou 10 no primeiro lance, essa soma torna-se sua "pontuação". Para ganhar, você deve continuar a rolar os dados até "fazer sua pontuação" (isto é, obter um valor igual à sua pontuação). Você perde se obtiver um 7 antes de fazer sua pontuação.*

```
1  // Figura 6.9: Craps.java
2  // A classe Craps simula o jogo de dados craps.
3  import java.util.Random;
4
5  public class Craps
6  {
7     // cria gerador de números aleatórios para uso no método rollDice
8     private static final Random randomNumbers = new Random();
9
10    // enumeração com constantes que representam o status do jogo
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constantes que representam lançamentos comuns dos dados
14    private static final int SNAKE_EYES = 2;
15    private static final int TREY = 3;
16    private static final int SEVEN = 7;
17    private static final int YO_LEVEN = 11;
18    private static final int BOX_CARS = 12;
19
```

Declara constantes para o status do jogo

Declara constantes representando lançamentos comuns dos dados

Figura 6.9 | A classe `Craps` simula o jogo de dados craps. (Parte 1 de 4.)

```
20      // joga uma partida de craps
21      public void play()
22      {
23         int myPoint = 0; // pontos se não ganhar ou perder na 1a. rolagem
24         Status gameStatus; // pode conter CONTINUE, WON ou LOST
25
26         int sumOfDice = rollDice(); // primeira rolagem dos dados
27
28         // determina o status do jogo e a pontuação com base no 1. lançamento
29         switch ( sumOfDice )
30         {
31            case SEVEN: // ganha com 7 no primeiro lançamento
32            case YO_LEVEN: // ganha com 11 no primeiro lançamento
33               gameStatus = Status.WON;
34               break;
35            case SNAKE_EYES: // perde com 2 no primeiro lançamento
36            case TREY: // perde com 3 no primeiro lançamento
37            case BOX_CARS: // perde com 12 no primeiro lançamento
38               gameStatus = Status.LOST;
39               break;
```

Variável que armazena o status do jogo

Rola os dados para iniciar o jogo

O jogador ganha no primeiro lançamento; configura `gameStatus` como `WON`

O jogador perde no primeiro lançamento; configura `gameStauts` como `LOST`

Figura 6.9 | A classe `Craps` simula o jogo de dados craps. (Parte 2 de 4.)

```
40            default: // não ganhou nem perdeu, então registra a pontuação
41               gameStatus = Status.CONTINUE; // jogo não terminou
42               myPoint = sumOfDice; // informa a pontuação
43               System.out.printf( "Point is %d\n", myPoint );
44               break; // opcional no final do switch
45         } // fim do switch
46
47         // enquanto o jogo não estiver completo
48         while (gameStatus == Status.CONTINUE ) // nem WON nem LOST
49         {
50            sumOfDice = rollDice(); // lança os dados novamente
51
52            // determina o status do jogo
53            if ( sumOfDice == myPoint ) // vitória por pontuação
54               gameStatus = Status.WON;
55            else
56               if ( sumOfDice == SEVEN ) // perde obtendo 7 antes de atingir a pontuação
57                  gameStatus = Status.LOST;
58         } // fim do while
59
```

O jogador nem ganha nem perde; configura gameSatus como CONTINUE

Faz um loop enquanto o jogo não termina

Rola os dados de novo

O usuário fez sua pontuação; configura gameStatus como WON

Deu 7; configura gameSatus como WON

Figura 6.9 | A classe Craps simula o jogo de dados craps. (Parte 3 de 4.)

```
60          // exibe uma mensagem ganhou ou perdeu
61          if (gameStatus == Status.WON )
62             System.out.println( "Player wins" );
63          else
64             System.out.println( "Player loses" );
65       } // fim do método play
66
67       // lança os dados, calcula a soma e exibe os resultados
68       public int rollDice()
69       {
70          // seleciona valores aleatórios do dado
71          int die1 = 1 + randomNumbers.nextInt( 6 ); // Primeiro lançamento do dado
72          int die2 = 1 + randomNumbers.nextInt( 6 ); // Segundo lançamento do dado
73
74          int sum = die1 + die2; // soma dos valores dos dados
75
76          // exibe os resultados desse lançamento
77          System.out.printf( "Player rolled %d + %d = %d\n",
78             die1, die2, sum );
79
80          return sum; // retorna a soma dos dados
81       } // fim do método rollDice
82    } // fim da classe Craps
```

Exibe a mensagem indicando se o usuário ganhou ou perdeu

Figura 6.9 | A classe `Craps` simula o jogo de dados craps. (Parte 4 de 4.)


```
1  // Figura 6.10: CrapsTest.java
2  // Aplicativo para testar a classe Craps.
3
4  public class CrapsTest
5  {
6     public static void main( String[] args )
7     {
8        Craps game = new Craps();
9        game.play(); // joga um jogo de craps
10    } // fim de main
11 } // fim da classe CrapsTest
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

Figura 6.10 | Aplicativo para testar a classe `Craps`. (Parte 1 de 2.)

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

Figura 6.10 | Aplicativo para testar a classe `Craps`. (Parte 2 de 2.)

- Notas:
  - Observe que `myPoint` é inicializada como `0` a fim de assegurar que o aplicativo compilará.
  - Se você não inicializar `myPoint`, o compilador emite um erro, porque `myPoint` não recebe um valor em cada `case` da instrução `switch` e, portanto, o programa poderia tentar utilizar `myPoint` antes de receber um valor.
  - `gameStatus` recebe um valor em cada `case` da instrução `switch` — portanto, é garantido que será inicializado antes de ser utilizado e não precisa ser inicializado.

- **Tipo `enum Status`**
  - Uma **enumeração**, na sua forma mais simples, declara um conjunto de constantes representado pelos identificadores.
  - Um tipo especial de classe que é introduzida pela palavra-chave **`enum`** e o nome de um tipo.
  - Chaves delimitam o corpo de uma declaração `enum`.
  - Dentro das chaves, há uma lista separada por vírgulas de **constantes de enumeração**, cada uma representando um valor único.
  - Os identificadores em uma `enum` devem ser únicos.
  - Variáveis do tipo `enum` podem receber somente as constantes declaradas na enumeração.

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 6.1

Utilize somente letras maiúsculas nos nomes das constantes de enumeração. Isso destaca as constantes e nos lembra de que as constantes de enumeração não são variáveis.

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 6.2

Usar constantes de enumeração (como `Status.WON`*,* `Status.LOST` *e* `Status.CONTINUE`*) em vez de valores literais (como 0, 1 e 2) torna os programas mais fáceis de ler e manter.*

- Por que algumas constantes não são definidas em constantes `enum`.
  - O Java não permite que um `int` seja comparado com uma constante de enumeração.
  - Infelizmente, o Java não fornece uma maneira fácil de converter um valor `int` em uma constante `enum` particular.
  - Pode-se traduzir uma constante `int` em uma `enum` com uma instrução `switch` separada.
  - Isso seria claramente complicado e não melhoraria a legibilidade do programa (derrotando assim o propósito do usar `enum`).

6.11 Escopo das declarações

- As declarações introduzem nomes que podem ser utilizados para referenciar essas entidades Java.
- O **escopo** de uma declaração é a parte do programa que pode referenciar a entidade declarada pelo seu nome.
  - Dizemos que essa entidade está "no escopo" para essa parte do programa.
- (Para mais informações sobre escopo, consulte *Java Language Specification, Section 6.3: Scope of a Declaration*
  - `java.sun.com/docs/books/jls/third_edition/html/names.html#103228`

- Regras de escopo básicas:
  - O escopo de uma declaração de parâmetro é o corpo do método em que a declaração aparece.
  - O escopo de uma declaração de variável local é do ponto em que a declaração aparece até o final desse bloco.
  - O escopo de uma declaração de variável local que aparece na seção de inicialização do cabeçalho de uma instrução `for` é o corpo da instrução `for` e as outras expressões no cabeçalho.
  - Um método ou escopo de campo é o corpo inteiro da classe.
- Qualquer bloco pode conter declarações de variável.
- Se uma variável local ou um parâmetro em um método tiver o mesmo nome de um campo da classe, o campo permanece "oculto" até que o bloco termine a execução — isso é chamado **sombreamento.**

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.10

Um erro de compilação ocorre quando uma variável local é declarada mais de uma vez em um método.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 6.3

Se você não puder escolher um nome conciso que expresse a tarefa de um método, seu método talvez tente realizar tarefas em demasia. Divida esse método em vários métodos menores.


```
1  // Figura 6.11: Scope.java
2  // A classe Scope demonstra os escopos de campo e de variável local.
3
4  public class Scope
5  {
6     // campo acessível para todos os métodos dessa classe
7     private int x = 1;
8
9     // método begin cria e inicializa a variável local x
10    // e chama os métodos useLocalVariable e useField
11    public void begin()
12    {
13       int x = 5; // variável local x do método sombreia o campo x
14
15       System.out.printf( "local x in method begin is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable tem uma variável local x
18       useField(); // useField utiliza o campo x da classe Scope
19       useLocalVariable(); // useLocalVariable reinicializa a variável local x
20       useField(); // campo x da classe Scope retém seu valor
21
22       System.out.printf( "\nlocal x in method begin is %d\n", x );
23    } // fim do método begin
```

Escopo de classe

Escopo de método

Figura 6.11 | A classe Scope demonstrando os escopos de um campo e de variáveis locais. (Parte 1 de 2.)

```
24
25       // cria e inicializa a variável local x durante cada chamada
26       public void useLocalVariable()
27       {
28          int x = 25; // inicializa toda vez que useLocalVariable é chamado
29
30          System.out.printf(
31             "\nlocal x on entering method useLocalVariable is %d\n", x );
32          ++x; // modifica a variável local x desse método
33          System.out.printf(
34            "local x before exiting method useLocalVariable is %d\n", x );
35       } // fim do método useLocalVariable
36
37       // modifica o campo x da classe Scope durante cada chamada
38       public void useField()
39       {
40          System.out.printf(
41             "\nfield x on entering method useField is %d\n", x );
42          x *= 10; // modifica o campo x da classe Scope
43          System.out.printf(
44             "field x before exiting method useField is %d\n", x );
45       } // fim do método useField
46    } // fim da classe Scope
```

Escopo de método

Utiliza variável de instância x

Figura 6.11 | A classe Scope demonstrando os escopos de um campo e de variáveis locais. (Parte 2 de 2.)


```
1  // Figura 6.12: ScopeTest.java
2  // Aplicativo para testar a classe Scope.
3
4  public class ScopeTest
5  {
6     // ponto de partida do aplicativo
7     public static void main( String[] args )
8     {
9        Scope testScope = new Scope();
10       testScope.begin();
11    } // fim de main
12 } // fim da classe ScopeTest
```

Figura 6.12 | Aplicativo para testar a classe Scope.

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

Figura 6.12 | Aplicativo para testar a classe Scope.

6.12 Sobrecarga de método

- **Sobrecarga de método.**
  - Métodos do mesmo nome declarados na mesma classe.
  - Deve ter diferentes conjuntos de parâmetros.
- O compilador Java seleciona o método adequado examinando o número, os tipos e a ordem dos argumentos na chamada.
- Utilizado para criar vários métodos com o mesmo nome que realizam as mesmas tarefas, ou tarefas semelhantes, mas sobre tipos diferentes ou números diferentes de argumentos.
- Valores literais inteiros são tratados como um tipo `int`, a chamada de método na linha 9 invoca a versão de `square` que especifica um parâmetro `int`.
- Valores de ponto flutuante são tratados como um tipo `double`, a chamada de método na linha 10 invoca a versão de `square` que especifica um parâmetro `int`.


```
1  // Figura 6.13: MethodOverload.java
2  // Declarações de métodos sobrecarregados.
3
4  public class MethodOverload
5  {
6     // teste de métodos square sobrecarregados
7     public void testOverloadedMethods()
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // fim do método testOverloadedMethods
12
13    // método square com argumento int
14    public int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // fim do método square com argumento int
20
```

Chama `square` com um parâmetro `int`

O método `square` recebe um `int`

Chama `square` com um parâmetro `double`

Figura 6.13 | Declarações de métodos sobrecarregados. (Parte 2 de 2.)


```
21      // método square com argumento double
22      public double square( double doubleValue )
23      {
24         System.out.printf( "\nCalled square with double argument: %f\n",
25            doubleValue );
26         return doubleValue * doubleValue;
27      } // fim do método square com argumento double
28   } // fim da classe MethodOverload
```

Método `square` que recebe um `double`

Figura 6.13 | Declarações de métodos sobrecarregados. (Parte 2 de 2.)

```
1  // Fig. 6.14: MethodOverloadTest.java
2  // Application to test class MethodOverload.
3
4  public class MethodOverloadTest
5  {
6     public static void main( String[] args )
7     {
8        MethodOverload methodOverload = new MethodOverload();
9        methodOverload.testOverloadedMethods();
10    } // end main
11 } // end class MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.14 | Application to test class `MethodOverload`.

6.12 Sobrecarga de método

- Distinguindo entre métodos sobrecarregados.
  - O compilador distingue métodos sobrecarregados por suas **assinaturas** — os nomes dos métodos e o número, os tipos e a ordem dos seus parâmetros.
- Tipos de retorno dos métodos sobrecarregados.
  - *Não é possível distinguir métodos pelo tipo retornado.*
- O programa na Figura 6.15 ilustra os erros de compilador quando dois métodos gerados têm a mesma assinatura e tipos diferentes de retorno.
- Métodos sobrecarregados podem ter diferentes tipos de retorno se os métodos tiverem diferentes listas de parâmetro.
- Métodos sobrecarregados não precisam ter o mesmo número de parâmetros.

Java™ COMO PROGRAMAR 8ª edição

Erro de programação comum 6.11

Declarar métodos sobrecarregados com listas de parâmetros idênticas é um erro de compilação independentemente de os tipos de retorno serem diferentes.

```
1  // Figura 6.15: MethodOverloadError.java
2  // Métodos sobrecarregados com assinaturas idênticas resultam
3  // em erros de compilação, mesmo que os tipos de retorno sejam diferentes.
4
5  public class MethodOverloadError
6  {
7     // declaração do método square com argumento int
8     public int square( int x )
9     {
10       return x * x;
11    }
12
13    // segunda declaração do método square com argumento int resulta
14    // em erros de compilação mesmo que os tipos de retorno sejam diferentes
15    public double square( int y )
16    {
17       return y * y;
18    }
19 } // fim da classe MethodOverloadError
```

Gera um erro de compilação

Figura 6.15 | Declarações de métodos sobrecarregados com assinaturas idênticas causam erros de compilação, mesmo que os tipos de retorno sejam diferentes. (Parte 1 de 2.)

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Figura 6.15 | Declarações de métodos sobrecarregados com assinaturas idênticas causam erros de compilação, mesmo que os tipos de retorno sejam diferentes. (Parte 2 de 2.)

6.13 (Opcional) Estudo de caso de GUI e imagens gráficas: cores e formas preenchidas

- As cores exibidas nas telas dos computadores são definidas pelos seus componentes vermelho, verde e azul (**valores RGB**) que têm valores inteiros de 0 a 255.
- Quanto mais alto o valor de um componente, mais rica intensa será essa tonalidade na cor final.
- O Java utiliza a classe **Color** no pacote `java.awt` para representar cores utilizando valores RGB.
- A classe `Color` contém 13 objetos `static Color` predefinidos — `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` e `YELLOW`.

- A classe `Color` também contém um construtor na forma:
  - `public Color( int r, int g, int b )`
- portanto, você pode criar cores personalizadas especificando os valores para os componentes vermelho, verde e azul individuais de uma cor.
- Os métodos `Graphics` **fillRect** e **fillOval** desenham retângulos e ovais preenchidos, respectivamente.
- O método `Graphics` **setColor** configura a cor atual do desenho.


```
1  // Figura 6.16: DrawSmiley.java
2  // Demonstra formas preenchidas.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7
8  {
9     public void paintComponent( Graphics g )
10    {
11       super.paintComponent( g );
12
13       // desenha o rosto
14       g.setColor( Color.YELLOW );
15       g.fillOval( 10, 10, 200, 200 );
16
17       // desenha os olhos
18       g.setColor( Color.BLACK );
19       g.fillOval( 55, 65, 30, 30 );
20       g.fillOval( 135, 65, 30, 30 );
21
22       // desenha a boca
23       g.fillOval( 50, 110, 120, 60 );
24
```

Figura 6.16 | Desenhando um rosto sorridente com cores e formas preenchidas. (Parte 1 de 2.)


```
25          // "retoca" a boca para criar um sorriso
26          g.setColor( Color.YELLOW );
27          g.fillRect( 50, 110, 120, 30 );
28          g.fillOval( 50, 120, 120, 40 );
29       } // fim do método paintComponent
30    } // fim da classe DrawSmiley
```

Figura 6.16 | Desenhando um rosto sorridente com cores e formas preenchidas. (Parte 2 de 2.)


```
1  // Figura 6.17: DrawSmileyTest.java
2  // Aplicativo de teste que exibe um rosto sorridente.
3  import javax.swing.JFrame;
4
5  public class DrawSmileyTest
6  {
7     public static void main( String[] args )
8     {
9        DrawSmiley panel = new DrawSmiley();
10       JFrame application = new JFrame();
11
12       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       application.add( panel );
14       application.setSize( 230, 250 );
15       application.setVisible( true );
16    } // fim de main
17 } // fim da classe DrawSmileyTest
```

Figura 6.17 | Criando `JFrame` para exibir um rosto sorridente. (Parte 1 de 2.)

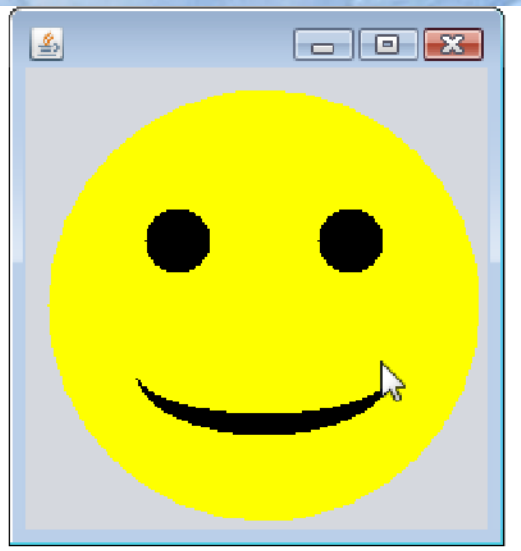


Figura 6.17 | Criando `JFrame` para exibir um rosto sorridente. (Parte 2 de 2.)