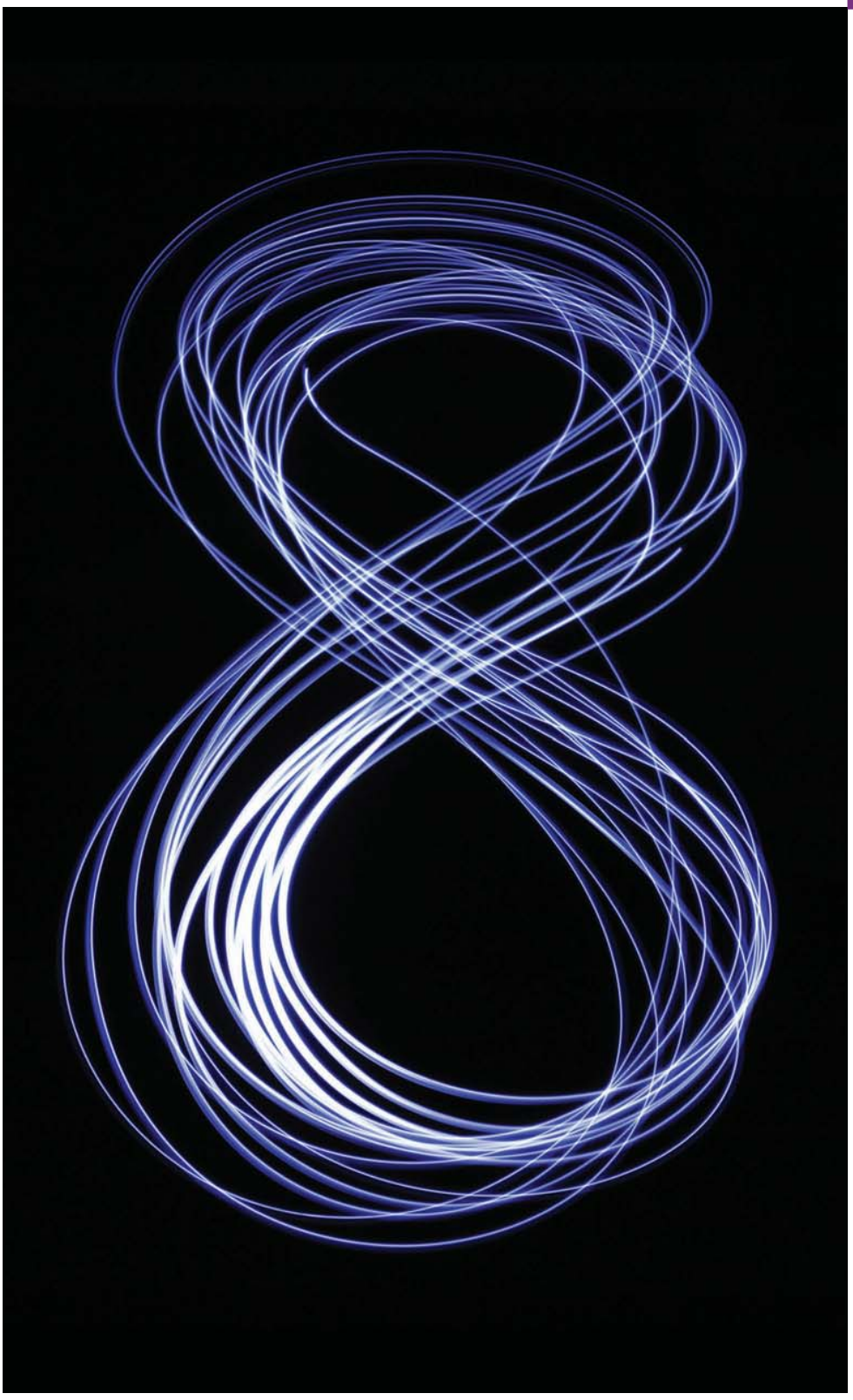
Métodos: um exame mais profundo

6



A forma nunca segue a função.
— Louis Henri Sullivan

E pluribus unum. (Um composto de muitos.)
— Virgílio

Chama o dia de ontem, faze que o tempo atrás retorne.
— William Shakespeare

Responda-me em uma palavra.
— William Shakespeare

Há um ponto em que os métodos se autodevoram.
— Frantz Fanon

Objetivos

Neste capítulo, você aprenderá:

- Como métodos e campos `static` se associam a classes em vez de objetos.
- Como o mecanismo de chamada/retorno de método é suportado pela pilha de chamadas de método.
- Sobre promoção e coerção de argumentos.
- Como pacotes agrupam classes relacionadas.
- Como utilizar a geração de números aleatórios seguros 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.

Sumário

6.1 Introdução

A experiência mostrou que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenos e simples pedaços, ou **módulos**. Essa técnica é chamada **dividir para conquistar**. Os métodos, que introduzimos no Capítulo 3, irão ajudá-lo a modularizar programas. Neste capítulo, estudamos os métodos em maior profundidade.

Você aprenderá mais sobre métodos `static` que podem ser chamados sem que um objeto da classe precise existir, e também como o Java é capaz de monitorar qual método está atualmente em execução, como variáveis locais dos métodos são mantidas na memória e como um método sabe para onde retornar depois de completar a execução.

Faremos uma breve digressão para examinar as técnicas de simulação com a geração de números aleatórios e desenvolveremos uma versão do jogo de dados de cassino chamado de craps, que utiliza a maioria das técnicas de programação usadas até agora neste livro. Além disso, você aprenderá a declarar constantes nos seus programas.

Boa parte das classes que você utilizará ou criará ao desenvolver aplicativos terá mais de um método com o mesmo nome. Essa técnica, chamada de *sobrecarregamento*, é utilizada para implementar os métodos que realizam tarefas semelhantes para argumentos de tipos diferentes ou para diferentes números de argumentos.

Continuaremos nossa discussão de métodos no Capítulo 18, "Recursão". A recursão fornece um modo intrigante de pensar nos métodos e algoritmos.

6.2 Módulos de programa em Java

Você escreve programas Java combinando novos métodos e classes com aqueles predefinidos disponíveis na **Java Application Programming Interface** (também chamada **Java API** ou **biblioteca de classes Java**) e em várias outras bibliotecas de classes. 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 na Seção 21.4.10. A Java API fornece uma rica coleção de classes predefinidas que contém métodos para realizar cálculos matemáticos comuns, manipulações de string, manipulações de caractere, operações de entrada/saída, operações de banco de dados, operações de rede, processamento de arquivo, verificação de erros etc.

Observação de engenharia de software 6.1

Familiarize-se com a rica coleção de classes e métodos fornecidos pela Java API (`http://docs.oracle.com/javase/7/docs/api/`). A Seção 6.8 fornece uma visão geral dos vários pacotes comuns. O Apêndice, em inglês, na Sala Virtual do livro, explica como navegar pela documentação da 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.

Dividir para conquistar com classes e métodos

Classes e métodos ajudam a modularizar um programa separando suas tarefas em unidades autocontidas. As instruções no corpo dos métodos são escritas apenas uma vez, permanecem ocultas de outros métodos e podem ser reutilizadas a partir de várias localizações em um programa.

Uma motivação para modularizar um programa em métodos e classes é a abordagem *dividir para conquistar*, que torna o desenvolvimento de programas mais gerenciável, construindo programas a partir de peças mais simples e menores. Outra é a **capacidade de reutilização de software** — o uso de classes e métodos existentes como blocos de construção para criar novos programas.

Frequentemente, você pode criar programas sobretudo a partir de classes e métodos existentes em vez de construir um código personalizado. Por exemplo, nos programas anteriores, não definimos como ler os dados a partir do teclado — o Java fornece essas capacidades nos métodos da classe `Scanner`. Uma terceira motivação é *evitar a repetição de código*. Dividir um programa em métodos e classes significativos torna o programa mais fácil de ser depurado e mantido.

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.

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.

Observação de engenharia de software 6.3

Se 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 menores.

Relação hierárquica entre chamadas de método

Como você sabe, um método é invocado por uma chamada de método, e quando o método chamado termina sua tarefa, ele retorna o controle e possivelmente um resultado para o chamador. Uma analogia a essa estrutura de programa é a 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 que o chefe saiba. Esse "ocultamento" dos detalhes de implementação promove a boa engenharia de software. A Figura 6.1 mostra o método `chefe` se comunicando com vários métodos trabalhadores de uma maneira hierárquica. O método `chefe` divide as responsabilidades entre os vários métodos `trabalhadores`. Aqui, `trabalhador1` atua como um "método chefe" para `trabalhador4` e `trabalhador5`.

Dica de prevenção de erro 6.2

Ao chamar um método que retorna um valor que indica se o método realizou sua tarefa com sucesso, certifique-se de verificar o valor de retorno desse método e, se esse método não foi bem-sucedido, lide com a questão de forma adequada.

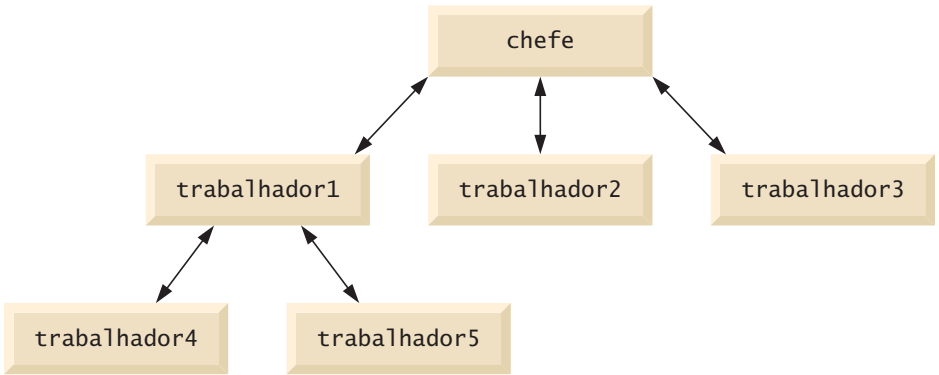


Figura 6.1 | Relacionamento hierárquico de método trabalhador/método chefe.

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

Embora a maioria dos métodos seja executada em resposta a chamadas de método *em objetos específicos*, esse nem sempre é o caso. Às vezes, um método realiza uma tarefa que não depende de um objeto. Esse método se aplica à classe em que é declarado como um todo e é conhecido como método `static` ou **método de classe**.

É comum que as classes contenham métodos `static` convenientes para realizar tarefas corriqueiras. Por exemplo, lembre-se de que utilizamos o método `static` `pow` da classe `Math` para elevar um valor a uma potência na Figura 5.6. Para declarar um método como `static`, coloque a palavra-chave `static` antes do tipo de retorno na declaração do método. Para qualquer classe importada para seu programa, você pode chamar métodos `static` da classe especificando o nome da classe na qual o método é declarado, seguido por um ponto (`.`) e nome do método, como em

```
NomeDaClasse.nomeDoMétodo(argumentos)
```

Os métodos da classe `Math`

Aqui, utilizamos vários métodos da classe `Math` para apresentar o conceito sobre os métodos `static`. A classe `Math` fornece uma coleção de métodos que permite realizar cálculos matemáticos comuns. Por exemplo, você pode calcular a raiz quadrada de `900.0` com a chamada do método `static`

```
Math.sqrt(900.0)
```

Essa expressão é avaliada como `30.0`. O método `sqrt` aceita um argumento do tipo `double` e retorna um resultado do tipo `double`. Para gerar a saída do valor da chamada do método anterior na janela de comando, você poderia escrever a instrução

```
System.out.println(Math.sqrt(900.0));
```

Nessa instrução, o valor que `sqrt` retorna torna-se o argumento ao método `println`. Não houve necessidade de criar um objeto `Math` antes de chamar o método `sqrt`. Além disso, *todos* os métodos da classe `Math` são `static` — portanto, cada um é chamado precedendo seu nome com o nome da classe `Math` e o ponto (`.`) separador.

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.

Os argumentos de método podem ser constantes, variáveis ou expressões. Se `c = 13.0`, `d = 3.0` e `f = 4.0`, então a instrução

```
System.out.println(Math.sqrt(c + d * f));
```

calcula e imprime a raiz quadrada de `13.0 + 3.0 * 4.0 = 25.0` — a saber `5.0`. A Figura 6.2 resume vários métodos da classe `Math`. Na figura, *x* e *y* são do tipo `double`.

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*`)`	cosseno 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`

continua

continuação

Método	Descrição	Exemplo
`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`
`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`.

Variáveis `static`

Lembre-se da Seção 3.2: cada objeto de uma classe mantém sua *própria* cópia de cada variável de instância da classe. Existem variáveis para as quais cada objeto de uma classe *não* precisa de sua própria cópia separada (como veremos mais adiante). Essas variáveis são declaradas `static`, também conhecidas como **variáveis de classe**. Quando os objetos de uma classe que contém variáveis `static` são criados, todos os objetos dessa classe compartilham *uma* cópia dessas variáveis. Juntas, variáveis `static` de uma classe e variáveis de instância são conhecidas como **campos**. Examinaremos outros detalhes sobre os campos `static` na Seção 8.11.

Constantes `PI` e `E` da classe `Math` `static`

A classe `Math` declara duas constantes, `Math.PI` e `Math.E`, que representam *aproximações de alta precisão* a constantes matemáticas comumente usadas. `Math.PI` (3,141592653589793) é a relação entre a circunferência de um círculo e seu diâmetro. `Math.E` (2,718281828459045) é o valor da base para logaritmos naturais (calculados com o método `static` `Math` `log`). Essas constantes são declaradas na classe `Math` com os modificadores `public`, `final` e `static`. Torná-los `public` permite que você os use 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. Tornar esses campos `static` permite que eles sejam acessados pelo nome da classe `Math` e um ponto (`.`) separador, como ocorre com os métodos da classe `Math`.

Por que o método `main` é declarado `static`?

Ao executar a Java Virtual Machine (JVM) com o comando `java`, a JVM tenta invocar o método `main` da classe que você especifica — neste ponto, quando nenhum objeto da classe tiver sido criado. Declarar `main` como `static` permite que a JVM invoque `main` sem criar uma instância da classe. Ao executar seu aplicativo, você especifica o nome da classe como um argumento para o comando `java`, como em

```
java NomeDaClasse argumento1 argumento2 ...
```

A JVM carrega a classe especificada pelo *NomeDaClasse* e utiliza esse nome para invocar o método `main`. No comando anterior, *NomeDaClasse* é um **argumento de linha de comando** para a JVM que informa qual classe executar. Depois do *NomeDaClasse*, você também pode especificar uma lista de `Strings` (separadas por espaços) como argumentos de linha de comando que a JVM passará para seu aplicativo. Esses argumentos poderiam ser utilizados para especificar opções (por exemplo, um nome de arquivo) a fim de executar o aplicativo. Como aprenderá no Capítulo 7, "Arrays e `ArrayLists`", seu aplicativo pode acessar esses argumentos de linha de comando e utilizá-los para personalizar o aplicativo.

6.4 Declarando métodos com múltiplos parâmetros

Os métodos costumam exigir mais de uma informação para realizar suas tarefas. Agora, iremos considerar como escrever seus próprios métodos com *múltiplos* parâmetros.

A Figura 6.3 utiliza um método chamado `maximum` para determinar e retornar o maior dos três valores `double`. No `main`, as linhas 14 a 18 solicitam que o usuário insira três valores `double`, então os lê a partir do usuário. A linha 21 chama o método `maximum` (declarado nas linhas 28 a 41) para determinar o maior dos três valores que recebe como argumentos. Quando o método `maximum` retorna o resultado para a linha 21, o programa atribui o valor de retorno de `maximum` à variável local `result`. Em seguida, a linha 24 gera a saída do valor máximo. No final desta seção, discutiremos o uso do operador + na linha 24.

```
1  // Figura 6.3: MaximumFinder.java
2  // Método maximum declarado pelo programador com três parâmetros double.
3  import java.util.Scanner;
4
```

continua

continuação

```
5  public class MaximumFinder
6  {
7     // obtém três valores de ponto flutuante e localiza o valor máximo
8     public static void main(String[] args)
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
23       // exibe o valor máximo
24       System.out.println("Maximum is: " + result);
25    }
26
27    // retorna o máximo dos seus três parâmetros de double
28    public static 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    }
42 } // fim da classe MaximumFinder
```

```
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 6.3 | O método declarado pelo programador `maximum` com três parâmetros `double`.

As palavras-chave `public` e `static`

A declaração do método `maximum` começa com palavra-chave `public` para indicar que o método está "disponível ao público" — ela pode ser chamada a partir dos métodos das outras classes. A palavra-chave `static` permite que o método `main` (outro método `static`) chame `maximum` como mostrado na linha 21 sem qualificar o nome do método com o nome da classe `MaximumFinder` — métodos `static` na mesma classe podem chamar uns aos outros diretamente. Qualquer outra classe que usa `maximum` deve qualificar totalmente o nome do método com o nome da classe.

Método `maximum`

Considere a declaração de `maximum` (linhas 28 a 41). A linha 28 indica que ele retorna um valor `double`, que o nome do método é `maximum` e que o método requer três parâmetros `double` (`x`, `y` e `z`) para realizar sua tarefa. Vários parâmetros são especificados como uma lista separada por vírgulas. Quando `maximum` é chamado a partir da linha 21, os parâmetros `x`, `y` e `z` são inicializados com cópias dos valores de argumentos `number1`, `number2` e `number3`, respectivamente. Deve haver um argumento na chamada de

método para cada parâmetro na declaração do método. Além disso, cada argumento deve ser *consistente* com o tipo do parâmetro correspondente. Por exemplo, um parâmetro do tipo `double` pode receber valores como 7.35, 22 ou –0.03456, mas não `String`s como `"hello"` nem valores `boolean` `true` ou `false`. A Seção 6.7 discute os tipos de argumento que podem ser fornecidos em uma chamada de método para cada parâmetro de um tipo primitivo.

Para determinar o valor máximo, começamos com a suposição de que o parâmetro `x` contém o maior valor, assim a linha 30 declara a variável local `maximumValue` e a inicializa com o valor do parâmetro `x`. Naturalmente, é possível que o parâmetro `y` ou `z` contenham o maior valor real, portanto devemos comparar cada um desses valores com `maximumValue`. A instrução `if` nas linhas 33 e 34 determina se `y` é maior que `maximumValue`. Se for, a linha 34 atribui `y` a `maximumValue`. A instrução `if` nas linhas 37 e 38 determina se `z` é maior que `maximumValue`. Se for, a linha 38 atribui `z` a `maximumValue`. Nesse ponto, o maior dos três valores reside em `maximumValue`, de modo que a linha 40 retorna esse valor à linha 21. Quando o controle de programa retornar ao ponto no programa em que `maximum` foi chamado, os parâmetros de `maximum` `x`, `y` e `z` não existirão mais na memória.

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.

Observação de engenharia de software 6.6

Variáveis devem ser declaradas como campos da 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.

Erro comum de programação 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`

O corpo inteiro do nosso método maximum também poderia ser implementado com duas chamadas a `Math.max`, como a seguir:

```
return Math.max(x, Math.max(y, z));
```

A primeira chamada para `Math.max` 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. Portanto, na instrução anterior, `Math.max(y, z)` é avaliada para determinar o valor máximo de `y` e `z`. O resultado é então passado como o segundo argumento para a outra chamada a `Math.max`, que retorna o maior dos seus dois argumentos. Esse é um bom exemplo da *reutilização de software* — encontramos o maior dos três valores reutilizando `Math.max`, que encontra o maior dos dois valores. Observe a concisão desse código comparado com as linhas 30 a 38 da Figura 6.3.

Montando strings com a concatenação de strings

O Java permite montar objetos `String` em strings maiores utilizando os operadores + ou +=. Isso é conhecido como **concatenação de strings**. Quando ambos os operandos do operador + são objetos `String`, o operador + cria um novo objeto `String` em que os caracteres do operando direito são colocados no fim daqueles no operando esquerdo — por exemplo, a expressão `"hello" + "there"` cria a `String` `"hello there"`.

Na linha 24 da Figura 6.3, a expressão `"Maximum is: " +` utiliza o operador `result` + com os operandos dos tipos `String` e `double`. *Cada objeto e valor primitivo no Java podem ser representados como uma `String`.* Se um dos operandos do operador + for uma `String`, o outro é convertido em uma `String` e então os dois são *concatenados*. Na linha 24, o valor de `double` é convertido na sua representação `String` e colocado no final da `String` `"Maximum is: "`. Um ou mais *zeros no final* de um valor `double` são *descartados* quando o número é convertido em uma `String` — por exemplo, 9.3500 seria representado como 9.35.

Valores primitivos usados na concatenação de `String` são convertidos em `String`s. Um `boolean` concatenado com uma `String` é convertido na `String` `"true"` ou `"false"`. *Todos os objetos têm um método `toString` que retorna uma representação `String` do objeto.* (Discutimos `toString` em mais detalhes em capítulos subsequentes.) Quando um objeto é concatenado com uma `String`, o método `toString` do objeto é chamado implicitamente para obter a representação de `String` do objeto. O método `toString` também pode ser chamado explicitamente.

Você pode dividir literais de `String` grandes em várias `Strings` menores e colocá-las em múltiplas linhas do código a fim de facilitar a leitura. Nesse caso, as `Strings` podem ser montadas utilizando a concatenação. Discutiremos os detalhes das `Strings` no Capítulo 14.

Erro comum de programação 6.2

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

Erro comum de programação 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 para a string `"y + 2 = "`, em seguida o valor 2 é concatenado para a 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

Há três maneiras de chamar um método:

1. Usando o próprio nome de método para chamar outro método da *mesma* classe, como maximum (`number1`, `number2`, `number3`) na linha 21 da Figura 6.3.
2. Utilizar uma variável que contém uma referência a um objeto, seguido por um ponto (`.`) e o nome do método para chamar um método não `static` do objeto referenciado — tal como a chamada de método na linha 16 da Figura 3.2, `myAccount.getName()`, que chama um método da classe `Account` a partir do método `main` de `AccountTest`. Métodos não `static` são normalmente chamados **métodos de instância**.
3. Utilizar o nome de classe e um ponto (`.`) para chamar um método `static` de uma classe — como `Math.sqrt(900.0)` na Seção 6.3.

Um método `static` pode chamar outros métodos `static` da mesma classe diretamente (isto é, usando o próprio nome do método) e manipular variáveis `static` na mesma classe diretamente. Para acessar os métodos de instância e as variáveis de instância da classe, um método `static` deve usar uma referência a um objeto da classe. Os métodos de instância podem acessar todos os campos (variáveis de instância e variáveis `static`) e métodos da classe.

Lembre-se de que métodos `static` relacionam-se com uma classe como um todo, enquanto métodos de instância estão associados a uma instância específica (objeto) da classe e podem manipular as variáveis de instância desse objeto. Muitos objetos de uma classe, cada um com *suas* próprias cópias das variáveis de instância, podem existir ao mesmo tempo. Suponha que um método `static` deva invocar um método de instância diretamente. Como o método `static` saberia quais variáveis de instância do objeto devem ser manipuladas? O que aconteceria se *nenhum* objeto da classe existisse no momento em que o método de instância fosse invocado? Assim, o Java *não* permite que um método `static` acesse diretamente as variáveis de instância e os métodos de instância da mesma classe.

Há três maneiras de retornar o controle à instrução que chama um método. Se o método não retornar um resultado, o controle retornará quando o fluxo do programa alcançar a chave direita de fechamento do método ou quando a instrução

```
return;
```

for executada. Se o método retornar um resultado, a instrução

```
return expressão;
```

avalia a *expressão* e então retorna o resultado ao chamador.

Erro comum de programação 6.4

Declarar um método fora do corpo de uma declaração de classe ou dentro do corpo de um outro método é um erro de sintaxe.

Erro comum de programação 6.5

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

Erro comum de programação 6.6

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* deve *conter uma instrução* `return`*, que retorna 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 Pilhas de chamadas de método e quadros de pilha

Para entender como o Java realiza chamadas de método, precisamos primeiro considerar uma estrutura de dados (isto é, a coleção de itens de dados relacionados) conhecida como **pilha**. Você pode pensar em uma pilha como análoga a uma pilha de pratos. Quando um prato é colocado na pilha, normalmente ele é colocado na parte superior (conhecido como **inserir** o prato na pilha). De maneira semelhante, quando um prato é removido da pilha, ele é normalmente removido da parte superior (conhecido como **retirar** o prato da pilha). As pilhas são conhecidas como estruturas de dados do tipo **último a entrar, primeiro a sair** (*last-in, first-out* — **LIFO**) — o *último* item inserido na pilha é o *primeiro* item que é removido da pilha.

Quando um programa *chama* um método, o método chamado deve saber *retornar* ao seu chamador, então o *endereço de retorno* do método chamador é *inserido* na **pilha de execução do método**. Se uma série de chamadas de método ocorre, os sucessivos endereços de retorno são empilhados na ordem último a entrar, primeiro a sair, de modo que cada método possa retornar para seu chamador.

A pilha de chamadas de método também contém a memória para as *variáveis locais* (incluindo os parâmetros de método) utilizadas em cada invocação de um método durante a execução de um programa. Esses dados, armazenados como parte da pilha das chamadas de método, são conhecidos como **quadro de pilha** (ou **registro de ativação**) da chamada de método. Quando uma chamada de método é feita, o quadro de pilha para essa chamada de método é *colocado* na pilha de chamadas de método. Quando o método retorna ao seu chamador, o quadro (ou registro de ativação) dessa chamada de método é *retirado* da pilha e essas variáveis locais não são mais conhecidas para o programa. Se uma variável local que contém uma referência a um objeto for a única variável no programa com uma referência a esse objeto, então, quando o quadro de pilha que contém essa variável local é removido da pilha, o objeto não pode mais ser acessado pelo programa e acabará por ser excluído da memória pela JVM durante a *coleta de lixo,* que discutiremos na Seção 8.10.

É claro que a memória de um computador é finita, portanto, apenas certa quantidade pode ser usada para armazenar quadros de pilha na pilha de chamadas de método. Se mais chamadas de método forem feitas do que o quadro de pilha pode armazenar, ocorre um erro conhecido como **estouro de pilha** — discutiremos isso com mais detalhes no Capítulo 11, "Tratamento de exceção: um exame mais profundo".

6.7 Promoção e coerção de argumentos

Um outro recurso importante das chamadas de método é a **promoção de argumentos** — converter um *valor do argumento*, se possível, no tipo que o método espera receber no seu *parâmetro* correspondente. Por exemplo, um programa pode chamar `sqrt` do método `Math` com um argumento `int`, embora um argumento `double` seja esperado. A instrução

```
System.out.println(Math.sqrt(4));
```

avalia corretamente `Math.sqrt(4)` e imprime o valor `2.0`. A lista de parâmetros da declaração de método faz com que o Java converta o valor `int 4` no valor `double 4.0` *antes* de passar o valor para o método `sqrt`. Essas conversões podem levar a erros de compilação se as **regras de promoção** do Java não forem satisfeitas. As regras especificam quais conversões são autorizadas — isto é, quais conversões podem ser realizadas *sem perda de dados*. No exemplo `sqrt` anterior, um `int` é convertido em um `double` sem alterar o seu valor. Entretanto, converter um `double` em um `int` *trunca* a parte fracionária do valor `double` — portanto, parte do valor é perdida. Converter tipos inteiros grandes em tipos inteiros pequenos (por exemplo, `long` em `int`, ou `int` em `short`) também pode resultar em valores alterados.

As regras de promoção se aplicam a expressões que contêm 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. Na verdade, a expressão utiliza uma *cópia temporária* de cada valor — os tipos dos valores originais permanecem inalterados. A Figura 6.4 lista os tipos primitivos e os tipos para os quais cada um pode ser promovido. As promoções válidas para um dado tipo sempre são para um tipo mais alto na tabela. Por exemplo, um `int` pode ser promovido aos tipos `long`, `float` e `double` mais altos.

Converter valores em tipos mais baixos na tabela da Figura 6.4 resultará em diferentes valores se o tipo mais baixo não puder representar o valor do tipo mais alto (por exemplo, o valor `int` 2000000 não pode ser representado como um `short` e qualquer número de ponto flutuante com dígitos depois do seu ponto de fração decimal não pode ser representado em um tipo inteiro como `long`, `int` ou `short`). Portanto, nos casos em que as informações podem ser perdidas por causa da conversão, o compilador Java requer que você utilize um *operador de coerção* (introduzido na Seção 4.10) para forçar explicitamente que a conversão ocorra — do contrário, ocorre um erro de compilação. Isso permite que você "assuma o controle" do compilador. Você essencialmente diz, "Sei que essa conversão poderia causar perda das informações, mas, aqui, para meus propósitos, isso não é um problema". Suponha que o método `square` calcule o quadrado de um inteiro e assim requeira um argumento `int`. Para chamarmos `square` com um argumento `double` chamado `doubleValue`, deveríamos escrever a chamada de método como

```
square((int) doubleValue)
```

Essa chamada de método faz uma coerção explícita (converte) do valor de `doubleValue` em um inteiro temporário para uso no método `square`. Assim, se o valor do `doubleValue` for `4.5`, o método receberá o valor `4` e retornará `16`, não `20.25`.

Erro comum de programação 6.7

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.

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 números em Java)

Figura 6.4 | Promoções permitidas para tipos primitivos.

6.8 Pacotes de Java API

Como vimos, o Java contém muitas classes *predefinidas* que são agrupadas em categorias de classes relacionadas chamadas de *pacotes*. Em conjunto, eles são conhecidos como a Java API (Java Application Programming Interface) ou a biblioteca de classes Java. Uma grande capacidade do Java são as milhares de classes da Java API. Alguns pacotes-chave da Java API que usamos neste livro estão descritos na Figura 6.5, que representam apenas uma pequena parte dos *componentes reutilizáveis* na Java API.

Pacote	Descrição
`java.awt.event`	O **Java Abstract Window Toolkit Event Package** contém classes e interfaces que permitem tratamento de evento para componentes GUI em nos pacotes `java.awt` e `javax.swing` (Veja o Capítulo 12, "Componentes GUI: parte 1", e o Capítulo 22, "Componentes GUI: parte 2".)
`java.awt.geom`	O **Java 2D Shapes Package** contém classes e interfaces para trabalhar com as avançadas capacidades gráficas bidimensionais do Java. (Veja o Capítulo 13, "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. (Veja o Capítulo 15, "Arquivos, fluxos e serialização de objetos".)
`java.lang`	O **Java Language Package** contém classes e interfaces (discutidas por todo este texto) que são exigidas por muitos programas Java. Esse pacote é importado pelo compilador em todos os programas.

continua

continuação

Pacote	Descrição
`java.net`	O **Java Networking Package** contém classes e interfaces que permitem aos programas comunicar-se via redes de computadores, como a internet. (Veja o Capítulo 28, em inglês, na Sala Virtual do livro.)
`java.security`	O **Java Security Package** contém classes e interfaces para melhorar a segurança do aplicativo.
`java.sql`	O **JDBC Package** contém classes e interfaces para trabalhar com bancos de dados. (Veja o Capítulo 24, "Acesso a bancos de dados com JDBC".)
`java.util`	O **Java Utilities Package** contém classes e interfaces utilitárias que permitem armazenar e processar grandes quantidades de dados. Muitas dessas classes e interfaces foram atualizadas para suportar novos recursos lambda do Java SE 8. (Veja o Capítulo 16, "Coleções genéricas".)
`java.util.concurrent`	O **Java Concurrency Package** contém classes utilitárias e interfaces para implementar programas que podem realizar múltiplas tarefas paralelamente. (Veja o Capítulo 23, "Concorrência".)
`javax.swing`	O **Java Swing GUI Components Package** contém classes e interfaces para componentes GUI Swing do Java que fornecem suporte para GUIs portáveis. Esse pacote ainda usa alguns elementos do pacote `java.awt` mais antigo. (Veja o Capítulo 12, "Componentes GUI: parte 1" e o Capítulo 22, "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`. (Veja o Capítulo 12, "Componentes GUI: parte 1" e o Capítulo 22, "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 32, em inglês, na Sala Virtual.)
pacotes `javafx`	JavaFX é a tecnologia da GUI preferida para o futuro. Discutiremos esses pacotes no Capítulo 25, "GUI do JavaFX: parte 1" e nos capítulos "JavaFX GUI" e "Multimídia", em inglês, na Sala Virtual.
Alguns pacotes Java SE 8 usados neste livro	
`java.time`	O novo **pacote** Java SE 8 **Date/Time API** contém classes e interfaces para trabalhar com datas e horas. Esses recursos são projetados para substituir as capacidades de data e hora mais antigas do pacote `java.util`. (Veja o Capítulo 23, "Concorrência".)
`java.util.function` e `java.util.stream`	Esses pacotes contêm classes e interfaces para trabalhar com as capacidades de programação funcional do Java SE 8. (Veja o Capítulo 17, "Lambdas e fluxos Java SE 8".)

Figura 6.5 | Pacotes da Java API (um subconjunto).

O conjunto de pacotes disponíveis no Java é bem grande. Além daqueles resumidos na Figura 6.5, o Java inclui pacotes para elementos gráficos complexos, interfaces de usuário gráficas avançadas, impressão, rede avançada, segurança, processamento de dados, multimídia, acessibilidade (para pessoas com deficiências), programação concorrente, criptografia, processamento XML e muitas outras capacidades. Para uma visão geral dos pacotes no Java, visite

```
http://docs.oracle.com/javase/7/docs/api/overview-summary.html
http://download.java.net/jdk8/docs/api/
```

Você pode localizar informações adicionais sobre os métodos de uma classe Java predefinida na documentação da Java API em

```
http://docs.oracle.com/javase/7/docs/api/
```

Ao visitar esse site, clique no link Index para ver uma listagem alfabética de todas as classes e métodos na Java API. Localize o nome da classe e clique no seu link para ver a descrição on-line dessa classe. Clique no link METHOD para ver uma tabela dos métodos da classe. Cada método `static` será listado com a palavra `static` antes do seu tipo de retorno.

6.9 Estudo de caso: geração segura de números aleatórios

Agora faremos uma divertida digressão para discutir um tipo popular de aplicativo de programação — simulação de jogos. Nesta seção, e na seção a seguir, desenvolveremos um programa bem estruturado de jogos com múltiplos métodos. Esse programa utiliza a maioria das instruções de controle apresentadas até aqui neste livro e introduz vários novos conceitos de programação.

O **elemento de acaso** pode ser introduzido em um programa por meio de um objeto da classe `SecureRandom` (pacote `java.security`). Esses objetos podem produzir valores aleatórios `boolean`, `byte`, `float`, `double`, `int`, `long` e gaussianos. Nos próximos vários exemplos, utilizaremos objetos da classe `SecureRandom` para produzir valores aleatórios.

Evoluindo para números aleatórios seguros

As edições mais recentes deste livro usavam a classe `Random` do Java para obter valores "aleatórios". Essa classe produzia valores *determinísticos* que poderiam ser *previstos* por programadores mal-intencionados. Objetos `SecureRandom` produzem **números aleatórios não determinísticos** que *não podem* ser previstos.

Números aleatórios determinísticos foram a fonte de muitas falhas de segurança de software. A maioria das linguagens de programação agora tem recursos de biblioteca semelhantes à classe `SecureRandom` do Java para gerar números aleatórios não determinísticos a fim de ajudar a evitar esses problemas. Desse ponto em diante no livro, quando nos referimos a "números aleatórios", queremos dizer "manter números aleatórios seguros".

Criando um objeto `SecureRandom`

Um novo objeto gerador seguro de números aleatórios pode ser criado como a seguir:

```
SecureRandom randomNumbers = new SecureRandom();
```

Ele pode então ser utilizado para gerar valores aleatórios — discutimos apenas valores `int` aleatórios aqui. Para obter mais informações sobre a classe `SecureRandom`, consulte `docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html`.

Obtendo um valor aleatório `int`

Considere a seguinte instrução:

```
int randomValue = randomNumbers.nextInt();
```

O método `SecureRandom` **`nextInt`** gera um valor aleatório `int`. Se ele realmente produzir valores *aleatoriamente*, então cada valor no intervalo deve ter uma *chance igual* (ou probabilidade) de ser escolhido toda vez que `nextInt` é chamado.

Alterando o intervalo de valores produzido por `nextInt`

O intervalo de valores produzido diretamente pelo método `nextInt` geralmente difere do intervalo de valores requerido em um aplicativo Java particular. Por exemplo, um programa que simula lançamento de moeda talvez requeira somente 0 para "caras" e 1 para "coroas". Um programa que simula o lançamento de um dado de seis faces exigiria inteiros aleatórios no intervalo 1 a 6. Um programa que prevê aleatoriamente o próximo tipo de espaçonave (entre quatro possibilidades) que voará pelo horizonte em um videogame exigiria inteiros aleatórios no intervalo 1 a 4. Para esses casos, a classe `SecureRandom` fornece uma outra versão do método `nextInt` que recebe um argumento `int` e retorna um valor a partir de 0, mas sem incluí-lo, até o valor do argumento. Por exemplo, para o lançamento de moedas, a seguinte instrução retorna 0 ou 1.

```
int randomValue = randomNumbers.nextInt(2);
```

Lançando um dado de seis faces

Para demonstrar números aleatórios, vamos desenvolver um programa que simula 20 lançamentos de um dado de seis faces e exibe o valor de cada lançamento. Iniciamos utilizando `nextInt` para produzir os valores aleatórios no intervalo de 0 a 5, como a seguir:

```
int face = randomNumbers.nextInt(6);
```

O argumento `6` — chamado de **fator de escalonamento** — representa o número de valores únicos que `nextInt` deve produzir (nesse caso, seis — 0, 1, 2, 3, 4 e 5). Essa manipulação é chamada **escalonar** o intervalo de valores produzido pelo método `SecureRandom` `nextInt`.

Um dado de seis lados tem os números 1 a 6 nas faces, não 0 a 5. Portanto, **deslocamos** o intervalo de números produzidos adicionando um **valor de deslocamento** — nesse caso, 1 — ao nosso resultado anterior, como em

```
int face = 1 + randomNumbers.nextInt(6);
```

O valor de deslocamento (`1`) especifica o *primeiro* valor no conjunto desejado de inteiros aleatórios. A instrução anterior atribui `face` a um inteiro aleatório no intervalo de 1 a 6.

Lançando um dado de seis faces 20 vezes

A Figura 6.6 mostra duas saídas de exemplo que confirmam o fato de que os resultados do cálculo anterior são inteiros no intervalo de 1 a 6, e que cada execução do programa pode produzir uma sequência *diferente* de números aleatórios. A linha 3 importa a classe `SecureRandom` do pacote `java.security`. A linha 10 cria o objeto `SecureRandom` `randomNumbers` para produzir valores aleatórios. A linha 16 executa 20 vezes em um loop para lançar o dado. A instrução `if` (linhas 21 e 22) no loop inicia uma nova linha de saída depois de cada cinco números para criar um formato de cinco colunas perfeito.

```
1  // Figura 6.6: RandomIntegers.java
2  // Inteiros aleatórios deslocados e escalonados.
3  import java.security.SecureRandom; // o programa usa a classe SecureRandom
4
5  public class RandomIntegers
6  {
7     public static void main(String[] args)
8     {
9        // o objeto randomNumbers produzirá números aleatórios seguros
10       SecureRandom randomNumbers = new SecureRandom();
11
12       // faz o loop 20 vezes
13       for (int counter = 1; counter <= 20; counter++)
14       {
15          // seleciona o inteiro aleatório entre 1 e 6
16          int 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       }
24    }
25 } // fim da classe RandomIntegers
```

```
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 6.6 | Inteiros aleatórios deslocados e escalonados.

Lançando um dado de seis faces 6.000.000 vezes

Para mostrar que os números produzidos por `nextInt` ocorrem com probabilidade aproximadamente igual, vamos simular 6.000.000 lançamentos de um dado com o aplicativo da Figura 6.7. Todo número inteiro de 1 a 6 deve aparecer cerca de 1.000.000 vezes.

```
1  // Figura 6.7: RollDie.java
2  // Rola um dado de seis lados 6.000.000 vezes.
3  import java.security.SecureRandom;
4
5  public class RollDie
6  {
7     public static void main(String[] args)
8     {
9        // o objeto randomNumbers produzirá números aleatórios seguros
10       SecureRandom randomNumbers = new SecureRandom();
11
12       int frequency1 = 0; // contagem de 1s lançados
13       int frequency2 = 0; // contagem de 2s lançados
14       int frequency3 = 0; // contagem de 3s lançados
15       int frequency4 = 0; // contagem de 4s lançados
16       int frequency5 = 0; // contagem de 5s lançados
17       int frequency6 = 0; // contagem de 6s lançados
18
```

continua

continuação

```
19          // soma 6.000.000 lançamentos de um dado
20          for (int roll = 1; roll <= 6000000; roll++)
21          {
22             int face = 1 + randomNumbers.nextInt(6); // número entre 1 e 6
23
24             // usa o valor 1-6 para as faces a fim de determinar qual contador incrementar
25             switch (face)
26             {
27                case 1:
28                   ++frequency1; // incrementa o contador de 1s
29                   break;
30                case 2:
31                   ++frequency2; // incrementa o contador de 2s
32                   break;
33                case 3:
34                   ++frequency3; // incrementa o contador de 3s
35                   break;
36                case 4:
37                   ++frequency4; // incrementa o contador de 4s
38                   break;
39                case 5:
40                   ++frequency5; // incrementa o contador de 5s
41                   break;
42                case 6:
43                   ++frequency6; // incrementa o contador de 6s
44                   break;
45             }
46          }
47
48          System.out.println("Face\tFrequency"); // cabeçalhos de saída
49          System.out.printf("1\t%d%n2\t%d%n3\t%d%n4\t%d%n5\t%d%n6\t%d%n",
50             frequency1, frequency2, frequency3, frequency4,
51             frequency5, frequency6);
52       }
53    } // fim da classe RollDie
```

```
Face    Frequency
1       999501
2       1000412
3       998262
4       1000820
5       1002245
6       998760
```

```
Face    Frequency
1       999647
2       999557
3       999571
4       1000376
5       1000701
6       1000148
```

Figura 6.7 | Rolando um dado de seis lados 6.000.000 vezes.

Como as saídas de exemplo mostram, escalonar e deslocar os valores produzidos pelo método `nextInt` permite que o programa simule de modo realista o lançamento de um dado de seis faces. O aplicativo utiliza as instruções de controle aninhadas (o `switch` é aninhado dentro do `for`) para determinar o número de vezes que a face do dado ocorreu. A instrução `for` (linhas 20 a 46) itera 6.000.000 vezes. Durante cada iteração, a linha 22 produz um valor aleatório de 1 a 6. Esse valor é então utilizado como a expressão de controle (linha 25) da instrução `switch` (linhas 25 a 45). Com base no valor de `face`, a instrução `switch` incrementa uma das seis variáveis de contador durante cada iteração do loop. Essa instrução `switch` não tem nenhum caso `default` porque há um `case` para cada valor possível do dado que a expressão na linha 22 pode produzir. Execute o programa e observe os resultados. Como você verá, toda vez que executar esse programa, ele produzirá resultados *diferentes*.

Ao estudar arrays no Capítulo 7, mostraremos uma maneira elegante de substituir toda a instrução `switch` nesse programa por uma *única* instrução. Então, quando estudarmos as novas capacidades de programação funcional do Java SE 8 no Capítulo 17, mostraremos como substituir o loop que lança os dados, a instrução `switch` *e* a instrução que exibe os resultados com uma *única* instrução!

Escalonamento e deslocamento generalizados de números aleatórios

Anteriormente, simulamos o lançamento de um dado de seis lados com a instrução

```
int face = 1 + randomNumbers.nextInt(6);
```

Essa instrução sempre atribui à variável `face` um inteiro no intervalo $1 \leq face \leq 6$. A *largura* desse intervalo (isto é, o número de inteiros consecutivos no intervalo) é `6` e o *número inicial* no intervalo é `1`. Na instrução anterior, a largura do intervalo é determinada pelo número `6`, que é passado como um argumento para o método `nextInt` `SecureRandom`, e o número inicial do intervalo é o número `1`, que é adicionado a `randomNumbers.nextInt(6)`. Podemos generalizar esse resultado como

```
int 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. Por exemplo, para obter um valor aleatório na sequência 2, 5, 8, 11 e 14, você poderia utilizar a instrução

```
int number = 2 + 3 * randomNumbers.nextInt(5);
```

Nesse caso, `randomNumbers.nextInt(5)` produz os valores do intervalo de 0 a 4. Cada valor produzido é multiplicado por 3 para produzir um número na sequência 0, 3, 6, 9 e 12. Adicionamos 2 a esse valor para *deslocar* o intervalo de valores e obter um valor da sequência 2, 5, 8, 11 e 14. Podemos generalizar esse resultado como

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

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

Uma nota sobre o desempenho

Usar `SecureRandom` em vez de `Random` para alcançar níveis mais altos de segurança causa uma penalidade de desempenho significativa. Para aplicativos "casuais", você pode querer usar a classe `Random` do pacote `java.util` — simplesmente substitua `SecureRandom` por `Random`.

6.10 Estudo de caso: um jogo de azar; apresentando tipos `enum`

Um jogo popular de azar é um jogo de dados conhecido como craps, que é jogado em cassinos e nas ruas de todo o mundo. As regras do jogo são simples e diretas:

> *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.*

A Figura 6.8 simula o jogo de dados, utilizando métodos para implementar a lógica do jogo. O método `main` (linhas 21 a 65) chama o método `rollDice` (linhas 68 a 81) como necessário para lançar os dados e calcular a soma deles. As saídas de exemplo demonstram a vitória e a derrota na primeira rolagem, e em uma rolagem subsequente.

```
1  // Figura 6.8: Craps.java
2  // A classe Craps simula o jogo de dados craps.
3  import java.security.SecureRandom;
4
5  public class Craps
6  {
7     // cria um gerador seguro de números aleatórios para uso no método rollDice
8     private static final SecureRandom randomNumbers = new SecureRandom();
```

continua

continuação

```
9
10       // tipo enum com constantes que representam o estado 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
20       // joga uma partida de craps
21       public static void main(String[] args)
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 estado do jogo e a pontuação com base no primeiro 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;
40             default: // não ganhou nem perdeu, portanto 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;
45          }
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 estado 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          }
59
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       }
66
67       // lança os dados, calcula a soma e exibe os resultados
68       public static 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
```

continua

continuação

```
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;
81       }
82    } // fim da classe Craps
```

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

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

```
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 6.8 | A classe `Craps` simula o jogo de dados craps.

Método `rollDice`

Nas regras do jogo, o jogador deve rolar *dois* dados na primeira e em todas as jogadas subsequentes. Declaramos o método `rollDice` (linhas 68 a 81) para lançar o dado, calcular e imprimir sua soma. O método `rollDice` é declarado uma vez, mas é chamado a partir de dois lugares (linhas 26 e 50) no método `main`, que contém a lógica para um jogo de *craps* completo. O método `rollDice` não recebe nenhum argumento, então tem uma lista vazia de parâmetro. Toda vez que é chamado, `rollDice` retorna a soma dos dados, assim o tipo de retorno `int` é indicado no cabeçalho do método (linha 68). Embora as linhas 71 e 72 pareçam idênticas (exceto quanto aos nomes dos dados), elas não necessariamente produzem o mesmo resultado. Cada uma dessas instruções produz um valor *aleatório* no intervalo de 1 a 6. A variável `randomNumbers` (utilizada nas linhas 71 e 72) *não* é declarada no método. Em vez disso, é declarada como uma variável `private static final` da classe e inicializada na linha 8. Isso permite criar um objeto `SecureRandom` que é reutilizado em cada chamada para `rollDice`. Se houvesse um programa que contivesse múltiplas instâncias da classe `Craps`, todas elas compartilhariam esse objeto `SecureRandom`.

Variáveis locais do método `main`

O jogo é razoavelmente complexo. O jogador pode ganhar ou perder na primeira rolagem ou ganhar ou perder em qualquer rolagem subsequente. O método `main` (linhas 21 a 65) utiliza a variável local `myPoint` (linha 23) para armazenar a "pontuação" se o jogador não ganhar nem perder no primeiro lançamento, a variável local `gameStatus` (linha 24) para monitorar o status geral do jogo e a variável local `sumOfDice` (linha 26) para manter a soma dos dados para o lançamento mais recente. A variável `myPoint` é inicializada como `0` para assegurar que o aplicativo irá compilar. Se você não inicializar `myPoint`, o compilador emite um erro, porque `myPoint` não recebe um valor em *cada* `case` da instrução `switch`, e assim o programa pode tentar utilizar `myPoint` antes de receber um valor. Por outro lado, `gameStatus` *recebe* um valor em *cada* `case` da instrução `switch` (incluindo o caso `default`) — portanto, garante-se que ele é inicializado antes de ser utilizado. Por isso, não precisamos inicializá-lo na linha 24.

Tipo `enum Status`

A variável local `gameStatus` (linha 24) é declarada como um novo tipo chamado `Status` (declarada na linha 11). O tipo `Status` é um membro `private` da classe `Craps`, porque `Status` será utilizado somente nessa classe. `Status` é um tipo chamado **tipo `enum`**, que, em sua forma mais simples, declara um conjunto de constantes representadas por identificadores. Um tipo `enum` é um tipo especial de classe que é introduzido pela palavra-chave `enum` e um nome de tipo (nesse caso, `Status`). Como com as classes, as

chaves delimitam o corpo de uma declaração `enum`. Entre as chaves há uma lista de **constantes `enum`** separadas por vírgulas, cada uma representando um valor único. Os identificadores em uma `enum` devem ser *únicos*. Você aprenderá mais sobre tipos `enum` no Capítulo 8.

Boa prática de programação 6.1

Use somente letras maiúsculas nos nomes das constantes `enum` para fazer com que eles se destaquem e o lembrem de que elas não são variáveis.

Variáveis do tipo `Status` podem receber somente as três constantes declaradas na enumeração (linha 11) ou ocorrerá um erro de compilação. Quando se ganha o jogo, o programa configura a variável local `gameStatus` como `Status.WON` (linhas 33 e 54). Quando se perde o jogo, o programa configura a variável local `gameStatus` como `Status.LOST` (linhas 38 e 57). Do contrário, o programa define a variável local `gameStatus` como `Status.CONTINUE` (linha 41) para indicar que o jogo ainda não acabou e os dados devem ser rolados novamente.

Boa prática de programação 6.2

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

Lógica do método `main`

A linha 26 no `main` chama `rollDice`, que seleciona dois valores aleatórios de 1 a 6, exibe os valores do primeiro dado, do segundo dado e a soma, e retorna a soma. Em seguida, o método `main` insere a instrução `switch` (linhas 29 a 45), que utiliza o valor `sumOfDice` na linha 26 para determinar se o jogo foi ganho ou perdido, ou se deve continuar com um outro lançamento. Os valores que resultam em uma vitória ou derrota na primeira jogada são declarados como constantes `private static final int` nas linhas 14 a 18. Os nomes do identificador usam jargão de cassino para essas somas. Essas constantes, como ocorre com constantes `enum`, são declaradas, por convenção, com todas as letras maiúsculas, fazendo-as se destacar no programa. As linhas 31 a 34 determinam se o jogador ganhou no primeiro lançamento com `SEVEN` (7) ou `YO_LEVEN` (11). As linhas 35 a 39 determinam se o jogador perdeu no primeiro lançamento com `SNAKE_EYES` (2), `TREY` (3) ou `BOX_CARS` (12). Depois do primeiro lançamento, se o jogo não tiver acabado, a opção `default` (linhas 40 a 44) configura `gameStatus` como `Status.CONTINUE`, salva `sumOfDice` em `myPoint` e exibe a pontuação.

Se ainda estivermos tentando "fazer nossa melhor pontuação" (isto é, o jogo continua a partir de um lançamento anterior), as linhas 48 a 58 são executadas. A linha 50 lança os dados novamente. Se `sumOfDice` coincidir com `myPoint` (linha 53), a linha 54 configura `gameStatus` como `Status.WON`, o loop termina porque o jogo está completo. Se `sumOfDice` for `SEVEN` (linha 56), a linha 57 configura `gameStatus` como `Status.LOST`, e o loop termina porque o jogo está completo. Quando o jogo é concluído, as linhas 61 a 64 exibem uma mensagem que indica se o jogador ganhou ou perdeu e o programa termina.

O programa utiliza vários mecanismos de controle de programa que discutimos. A classe `Craps` usa dois métodos — `main` e `rollDice` (chamado duas vezes a partir de `main` — e as instruções de controle `switch`, `while`, `if...else` e `if` aninhada. Observe também o uso de múltiplos rótulos `case` na instrução `switch` para executar as mesmas instruções para somas de `SEVEN` e `YO_LEVEN` (linhas 31 e 32) e para somas de `SNAKE_EYES`, `TREY` e `BOX_CARS` (linhas 35 a 37).

Por que algumas constantes não são definidas como constantes `enum`

Você poderia estar questionando por que declaramos as somas dos dados como constantes `private static final int` em vez de constantes `enum`. A razão é que o programa tem de comparar a variável `int sumOfDice` (linha 26) com essas constantes para determinar o resultado de cada jogada. Suponha que declaramos `enum Sum` contendo constantes (por exemplo, `Sum.SNAKE_EYES`) que representam as cinco somas utilizadas no jogo e, então, usamos essas constantes na instrução `switch` (linhas 29 a 45). Fazer isso evitaria a utilização de `sumOfDice` como a expressão de controle da instrução, `switch` — porque o Java *não* permite que um `int` seja comparado com uma constante de enumeração. Para conseguir a mesma funcionalidade do programa atual, teríamos de utilizar uma variável `currentSum` do tipo `Sum` como a expressão de controle do `switch`. Infelizmente, o Java não fornece uma maneira fácil de converter um valor `int` em uma constante `enum` particular. Isso pode ser feito com uma instrução `switch` separada. Isso seria complicado e não melhoraria a legibilidade do programa (destruindo assim o propósito do uso de um `enum`).

6.11 Escopo das declarações

Você viu declarações de várias entidades Java como classes, métodos, variáveis e parâmetros. 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. Diz-se que essa entidade está "no escopo" para essa parte do programa. Esta seção introduz várias questões importantes de escopo.

As regras básicas de escopo são estas:

1. O escopo de uma declaração de parâmetro é o corpo do método em que a declaração aparece.
2. O escopo de uma declaração de variável local é do ponto em que a declaração aparece até o final desse bloco.
3. 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.
4. O escopo de um método ou campo é o corpo inteiro da classe. Isso permite que os métodos de instância de uma classe usem os campos e outros métodos 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 de **sombreamento**. Para acessar um campo sombreado em um bloco:

- Se o campo é uma variável de instância, preceda o nome com a palavra-chave `this` e um ponto (`.`), como em `this.x`.
- Se o campo é uma variável de classe `static`, preceda o nome com o nome da classe e um ponto (`.`), como em *NomeDaClasse*`.x`.

A Figura 6.9 demonstra os problemas de escopo com campos e variáveis locais. A linha 7 declara e inicializa o campo `x` para `1`. Esse campo permanece *sombreado* (oculto) em qualquer bloco (ou método) que declara uma variável local chamada `x`. O método `main` (linhas 11 a 23) declara uma variável local `x` (linha 13) e a inicializa para `5`. O valor dessa variável local é gerado para mostrar que o campo `x` (cujo valor é `1`) permanece *sombreado* no método `main`. O programa declara outros dois métodos — `useLocalVariable` (linhas 26 a 35) e `useField` (linhas 38 a 45) — que não recebem argumentos e não retornam resultados. O método `main` chama cada método duas vezes (linhas 17 a 20). O método `useLocalVariable` declara a variável local `x` (linha 28). Quando `useLocalVariable` é chamado pela primeira vez (linha 17), ele cria a variável local `x` e a inicializa como `25` (linha 28), gera a saída do valor de `x` (linhas 30 e 31), incrementa `x` (linha 32) e gera a saída do valor de `x` novamente (linhas 33 e 34). Quando `uselLocalVariable` é chamado uma segunda vez (linha 19), ele *recria* a variável local `x` e a *reinicializa* como `25`, assim a saída de cada chamada a `useLocalVariable` é idêntica.

```
1  // Figura 6.9: 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 static int x = 1;
8
9     // O método main cria e inicializa a variável local x
10    // e chama os métodos useLocalVariable e useField
11    public static void main(String[] args)
12    {
13       int x = 5; // variável local x do método sombreia o campo x
14
15       System.out.printf("local x in main 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 main is %d%n", x);
23    }
24
25    // cria e inicializa a variável local x durante cada chamada
26    public static void useLocalVariable()
27    {
28       int x = 25; // inicializada toda vez que useLocalVariable é chamado
29
```

continua

continuação

```
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       }
36
37       // modifica o campo x da classe Scope durante cada chamada
38       public static 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       }
46    } // fim da classe Scope
```

```
local x in main 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 main is 5
```

Figura 6.9 | A classe `Scope` demonstra escopos de campo e de variável local.

O método `useField` não declara nenhuma variável local. Portanto, quando ele se refere a `x`, é o campo `x` (linha 7) da classe que é utilizado. Ao ser chamado pela primeira vez (linha 18), o método `useField` gera saída do valor (`1`) do campo `x` (linhas 40 e 41), multiplica o campo `x` por `10` (linha 42) e gera a saída do valor (`10`) do campo `x` novamente (linhas 43 e 44) antes de retornar. A próxima vez que o método `useField` é chamado (linha 20), o campo contém seu valor modificado (`10`), assim o método gera saída de `10` e então `100`. Por fim, no método `main`, o programa gera saída do valor da variável local `x` novamente (linha 22) para mostrar que nenhum método chama a variável local `x` do `main` modificado, pois todos os métodos se referiram às variáveis identificadas como `x` nos outros escopos.

Princípio do menor privilégio

Em um sentido geral, "coisas" devem ter as capacidades de que precisamos para fazer o trabalho, mas não mais. Um exemplo é o escopo de uma variável. Uma variável não deve ser visível quando ela não é necessária.

Boa prática de programação 6.3

Declare as variáveis o mais próximo possível de onde elas foram usadas pela primeira vez.

6.12 Sobrecarga de método

Os métodos com o *mesmo* nome podem ser declarados na mesma classe, contanto que tenham *diferentes* conjuntos de parâmetros (determinados pelo número, tipos e ordem dos parâmetros) — isso é chamado de **sobrecarga de métodos**. Quando um método sobrecarregado é chamado, o compilador Java seleciona o método adequado examinando o número, os tipos e a ordem dos argumentos na chamada. A sobrecarga de métodos é comumente utilizada 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. Por exemplo, os métodos `Math` `abs`, `min` e `max` (resumidos na Seção 6.3) são sobrecarregados com quatro versões:

1. Uma com dois parâmetros `double`.
2. Uma com dois parâmetros `float`.
3. Uma com dois parâmetros `int`.
4. Uma com dois parâmetros `long`.

Nosso próximo exemplo demonstra como declarar e invocar métodos sobrecarregados. Demonstraremos construtores sobrecarregados no Capítulo 8.

Declarando métodos sobrecarregados

A classe `MethodOverload` (Figura 6.10) inclui duas versões sobrecarregadas do método `square` — uma que calcula o quadrado de um `int` (e retorna um `int`) e uma que calcula o quadrado de um `double` (e retorna um `double`). Embora esses métodos tenham o mesmo nome e listas e corpos semelhantes de parâmetros, você pode pensar neles simplesmente como *diferentes* métodos. Talvez ajude pensar nos nomes dos métodos como "`square` de `int`" e "`square` de `double`", respectivamente.

A linha 9 invoca o método `square` com o argumento `7`. Valores literais inteiros são tratados como um tipo `int`, assim a chamada de método na linha 9 invoca a versão de `square` nas linhas 14 a 19, que especifica um parâmetro `int`. De maneira semelhante, a linha 10 invoca o método `square` com o argumento `7.5`. Valores de ponto flutuante literais são tratados como um tipo `double`, dessa forma a chamada de método na linha 10 invoca a versão de `square` nas linhas 22 a 27, que especifica um parâmetro `double`. Cada método primeiro gera a saída de uma linha de texto para provar que o método adequado foi chamado em cada caso. Os valores nas linhas 10 e 24 são exibidos com o especificador de formato `%f`. Não especificamos uma precisão em nenhum dos casos. Por padrão, valores de ponto flutuante são exibidos com seis dígitos de precisão se a precisão *não* for especificada no especificador de formato.

```
1  // Figura 6.10: MethodOverload.java
2  // Declarações de métodos sobrecarregados.
3
4  public class MethodOverload
5  {
6     // teste de métodos square sobrecarregados
7     public static void main(String[] args)
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    }
12
13    // método square com argumento de int
14    public static int square(int intValue)
15    {
16       System.out.printf("%nCalled square with int argument: %d%n",
17          intValue);
18       return intValue * intValue;
19    }
20
21    // método square com argumento double
22    public static double square(double doubleValue)
23    {
24       System.out.printf("%nCalled square with double argument: %f%n",
25          doubleValue);
26       return doubleValue * doubleValue;
27    }
28 } // fim da classe MethodOverload
```

```
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
```

Figura 6.10 | Declarações de métodos sobrecarregados.

Distinguindo entre métodos sobrecarregados

O compilador distingue os métodos sobrecarregados pelas suas **assinaturas** — uma combinação do *nome* e o *número* do método, *tipos* e *ordens* de seus parâmetros, mas *não* de seu tipo de retorno. Se o compilador examinasse somente os nomes do método durante a compilação, o código na Figura 6.10 seria ambíguo — o compilador não saberia distinguir entre os dois métodos `square` (linhas 14 a 19 e 22 a 27). Internamente, o compilador utiliza nomes de método mais longos, que incluem o nome original do método, os tipos de cada parâmetro e a ordem exata dos parâmetros para determinar se os métodos em uma classe são *únicos* nela.

Por exemplo, na Figura 6.10, o compilador utilizaria (internamente) o nome lógico "`square` de `int`" para o método `square` que especifica um parâmetro `int` e "`square` de `double`" para o método `square` que especifica um parâmetro `double` (os nomes reais que o compilador utiliza são mais confusos). Se a declaração do `method1` iniciar como

```
void method1(int a, float b)
```

o compilador então poderia utilizar o nome lógico "`method1` de `int` e `float`". Se os parâmetros forem especificados como

```
void method1(float a, int b)
```

o compilador então poderia utilizar o nome lógico "`method1` de `float` e `int`". A *ordem* dos tipos de parâmetros é importante — o compilador considera os dois cabeçalhos do `method1` anterior como sendo *distintos*.

Tipos de retorno dos métodos sobrecarregados

Na discussão sobre os nomes lógicos dos métodos utilizados pelo compilador, não mencionamos os tipos de retorno dos métodos. *As chamadas de método não podem ser distinguidas pelo tipo de retorno*. Se você tivesse sobrecarregado métodos que se diferenciassem *apenas* por seus tipos de retorno e chamasse um dos métodos em uma instrução autônoma como em:

```
square(2);
```

o compilador *não* seria capaz de determinar a versão do método a chamar, porque o valor de retorno é *ignorado*. Quando dois métodos têm a *mesma* assinatura e retornam tipos *diferentes*, o compilador emite uma mensagem de erro indicando que o método já está definido na classe. Métodos sobrecarregados *podem* ter *diferentes* tipos de retorno se os métodos tiverem *diferentes* listas de parâmetro. Além disso, métodos sobrecarregados *não* precisam ter o mesmo número de parâmetros.

Erro comum de programação 6.8

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

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

Embora você possa criar muitos designs interessantes apenas com linhas e formas básicas, a classe `Graphics` fornece várias outras capacidades. Os próximos dois recursos que introduzimos são cores e formas preenchidas. Acrescentar cor enriquece os desenhos que um usuário vê na tela do computador. As formas podem ser preenchidas com cores sólidas.

As cores exibidas na tela dos computadores são definidas pelos componentes *vermelho*, *verde* e *azul* (chamados de **valores RGB**) que têm valores inteiros de 0 a 255. Quanto mais alto o valor de uma cor componente, mais rica será a tonalidade na cor final. O Java usa a classe **`Color`** (pacote `java.awt`) para representar as cores utilizando valores RGB. Por conveniência, a classe `Color` contém vários objetos `static` `Color` predefinidos — `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` e `YELLOW`. Cada objeto pode ser acessado por meio do nome da classe e um ponto (`.`) como em `Color.RED`. Você pode criar cores personalizadas passando os valores dos componentes de vermelho, verde e azul para construtor da classe `Color`:

```
public Color(int r, int g, int b)
```

Os métodos **`fillRect`** e **`fillOval`** da classe `Graphics` desenham ovais e retângulos preenchidos, respectivamente. Esses métodos têm os mesmos parâmetros que `drawRect` e `drawOval`; os dois primeiros são as coordenadas do *canto superior esquerdo* da forma, enquanto os dois seguintes determinam a *largura* e a *altura*. O exemplo nas figuras 6.11 e 6.12 demonstra cores e formas preenchidas desenhando e exibindo um rosto amarelo sorridente na tela.

```
1  // Figura 6.11: DrawSmiley.java
2  // Desenhando um rosto sorridente com cores e formas preenchidas.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DrawSmiley extends JPanel
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
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    }
30 } // fim da classe DrawSmiley
```

Figura 6.11 | Desenhando um rosto sorridente com cores e formas preenchidas.

As instruções `import` nas linhas 3 a 5 da Figura 6.11 importam as classes `Color`, `Graphics` e `JPanel`. A classe `DrawSmiley` (linhas 7 a 30) utiliza a classe `Color` para especificar as cores do desenho, e utiliza a classe `Graphics` para desenhar.

Mais uma vez, a classe `JPanel` fornece a área em que desenhamos. A linha 14 no método `paintComponent` usa o método `setColor` de `Graphics` para definir a cor atual do desenho como `Color.YELLOW`. O método `setColor` requer um argumento `Color` para configurar a cor de desenho. Nesse caso, utilizamos o objeto predefinido `Color.YELLOW`.

A linha 15 desenha um círculo com um diâmetro de 200 para representar o rosto — se os argumentos de largura e altura forem idênticos, o método `fillOval` desenhará um círculo. Em seguida, a linha 18 configura a cor como `Color.Black` e as linhas 19 e 20 desenham os olhos. A linha 23 desenha a boca como uma oval, mas isso não é bem o que queremos.

Para criar um rosto feliz, vamos retocar a boca. A linha 26 configura a cor como `Color.YELLOW`, portanto quaisquer formas que desenhamos serão combinadas com o rosto. A linha 27 desenha um retângulo com metade da altura da boca. Isso apaga a metade superior da boca, deixando somente a metade inferior. Para criar um sorriso melhor, a linha 28 desenha uma outra oval para cobrir levemente a parte superior da boca. A classe `DrawSmileyTest` (Figura 6.12) cria e exibe um `JFrame` contendo o desenho. Quando o `JFrame` é exibido, o sistema chama o método `paintComponent` para desenhar o rosto sorridente.

```
1  // Figura 6.12: 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    }
17 } // fim da classe DrawSmileyTest
```

continua

continuação

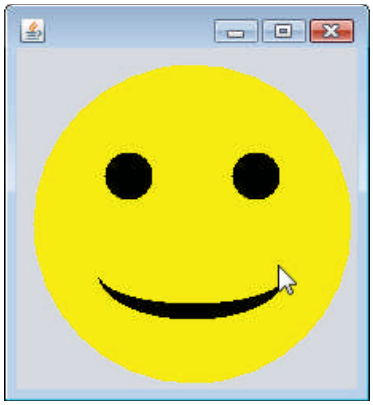


Figura 6.12 | Aplicativo de teste que exibe um rosto sorridente.

Exercícios do estudo de caso sobre GUIs e imagens gráficas

6.1 Utilizando o método `fillOval`, desenhe um alvo que alterna entre duas cores aleatórias, como na Figura 6.13. Utilize o construtor `Color(int r, int g, int b)` com argumentos aleatórios para gerar cores aleatórias.

6.2 Crie um programa que desenhe 10 formas preenchidas aleatórias com cores aleatórias, posições e tamanhos (Figura 6.14). `Method paintComponent` deve conter um loop que itera 10 vezes. Em cada iteração, o loop deve determinar se deve ser desenhado um retângulo ou uma oval preenchida, criar uma cor aleatória e escolher as coordenadas e dimensões aleatoriamente. As coordenadas devem ser escolhidas com base na largura e altura do painel. O comprimento dos lados deve estar limitado à metade da largura ou altura da janela.

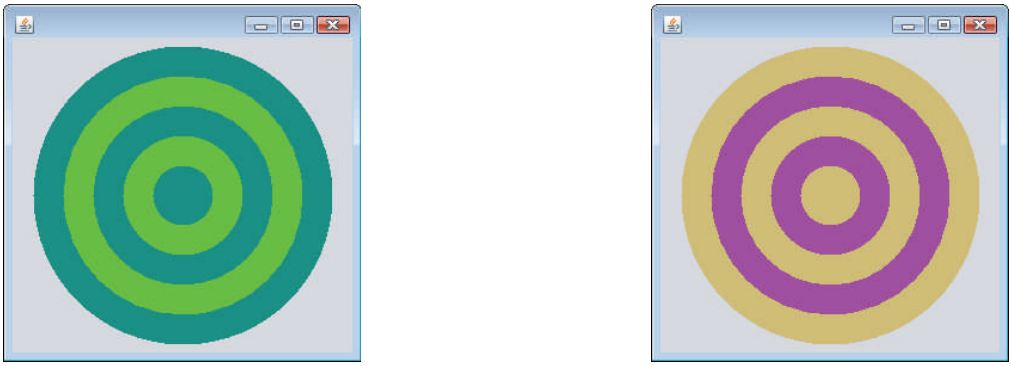


Figura 6.13 | Um alvo com duas cores aleatórias alternativas.

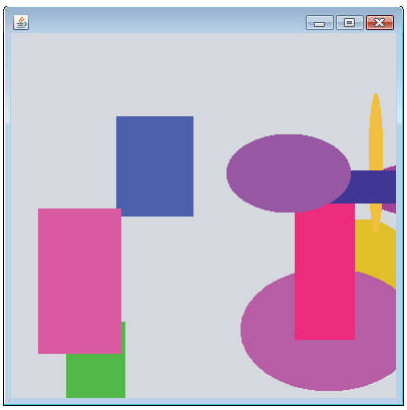


Figura 6.14 | Formas geradas aleatoriamente.

6.14 Conclusão

Neste capítulo, você aprendeu mais sobre declarações de método. Você também aprendeu a diferença entre métodos de instâncias e os métodos `static` e como chamar métodos `static` precedendo o nome do método com o nome da classe em que ele aparece e um ponto (`.`) separador. Aprendeu a utilizar os operadores `+` e `+=` para realizar concatenações de string. Discutimos como a pilha de chamadas de método e os registros de ativação monitoram os métodos que foram chamados e para onde cada método deve retornar quando ele completa a sua tarefa. Também discutimos as regras de promoção do Java para converter implicitamente entre tipos primitivos e como realizar conversões explícitas com operadores de coerção. Em seguida, você aprendeu sobre alguns dos pacotes mais utilizados na Java API.

Você aprendeu a declarar constantes identificadas utilizando tanto tipos `enum` como variáveis `private static final`. Você utilizou a classe `SecureRandom` para gerar números aleatórios para simulações. Você também aprendeu o escopo dos campos e variáveis locais em uma classe. Por fim, você aprendeu que múltiplos métodos em uma classe podem ser sobrecarregados fornecendo ao método o mesmo nome e assinaturas diferentes. Esses métodos podem ser utilizados para realizar as mesmas tarefas, ou tarefas semelhantes, utilizando tipos diferentes ou números distintos de parâmetros.

No Capítulo 7, você aprenderá a manter listas e tabelas de dados em arrays. Veremos uma implementação mais elegante do aplicativo que rola um dado 6.000.000 vezes. Apresentaremos duas versões de um estudo de caso `GradeBook` que armazena conjuntos das notas de alunos em um objeto `GradeBook`. Você também aprenderá a acessar os argumentos de linha de comando de um aplicativo que são passados para o método `main` quando um aplicativo começa a execução.

Resumo

Seção 6.1 Introdução

- A experiência mostrou que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenos e simples pedaços, ou módulos. Essa técnica é chamada dividir para conquistar.

Seção 6.2 Módulos de programa em Java

- Os métodos são declarados dentro de classes. Em geral, as classes são agrupadas em pacotes para que possam ser importadas e reutilizadas.
- Os métodos permitem modularizar um programa separando suas tarefas em unidades autocontidas. As instruções em um método são escritas somente uma vez e permanecem ocultas de outros métodos.
- Usar os métodos existentes como blocos de construção para criar novos programas é uma forma de reutilização de software que permite evitar repetição de código dentro de um programa.

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

- Uma chamada de método especifica o nome do método a ser chamado e fornece os argumentos que o método chamado requer para realizar sua tarefa. Quando a chamada de método é concluída, o método retorna um resultado, ou simplesmente o controle, ao seu chamador.
- Uma classe pode conter métodos `static` para realizar tarefas comuns que não exigem um objeto da classe. Quaisquer dados que um método `static` poderia requerer para realizar suas tarefas podem ser enviados ao método como argumentos em uma chamada de método. Um método `static` é chamado especificando o nome da classe em que o método é declarado seguido por um ponto (`.`) e pelo nome do método, como em

  NomeDaClasse`.`*nomeDoMétodo*`(`*argumentos*`)`

- A classe `Math` fornece os métodos `static` para realizar cálculos matemáticos comuns.
- A constante `Math.PI` (3,141592653589793) é a relação entre a circunferência de um círculo e seu diâmetro. A constante `Math.E` (2,718281828459045) é o valor base para logaritmos naturais (calculados com o método `static` `Math` `log`).
- `Math.PI` e `Math.E` são declaradas com os modificadores `public`, `final` e `static`. Torná-los `public` permite que você use esses campos nas suas próprias classes. Um campo declarado com a palavra-chave `final` é constante — seu valor não pode ser alterado depois de ele ser inicializado. Tanto `PI` como `E` são declarados `final` porque seus valores nunca mudam. Tornar esses campos `static` permite que eles sejam acessados pelo nome da classe `Math` e um ponto (`.`) separador, como ocorre com os métodos da classe `Math`.
- Todos os objetos de uma classe compartilham uma cópia dos campos `static` da classe. As variáveis de classe e as variáveis de instância representam os campos de uma classe.
- Quando você executa a Java Virtual Machine (JVM) com o comando `java`, a JVM carrega a classe especificada e utiliza esse nome de classe para invocar o método `main`. É possível especificar argumentos de linha de comando adicionais que a JVM passará para o seu aplicativo.
- Você pode colocar um método `main` em cada classe que declara — somente o método `main` na classe que você utiliza para executar o aplicativo será chamado pelo comando `java`.

Seção 6.4 Declarando métodos com múltiplos parâmetros

- Quando um método é chamado, o programa faz uma cópia dos valores de argumento do método e os atribui aos parâmetros correspondentes do método. Quando o controle do programa retorna ao ponto em que método foi chamado, os parâmetros do método são removidos da memória.
- Um método pode retornar no máximo um valor, mas o valor retornado poderia ser uma referência a um objeto que contém muitos valores.
- 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.
- Se um método tiver mais de um parâmetro, os parâmetros serão especificados como uma lista separada por vírgulas. Deve haver um argumento na chamada de método para cada parâmetro na declaração do método. Além disso, cada argumento deve ser consistente com o tipo do parâmetro correspondente. Se um método não aceitar argumentos, a lista de parâmetros ficará vazia.
- `Strings` podem ser concatenadas com o operador +, o que posiciona os caracteres do operando direito no final daqueles no operando esquerdo.
- Cada objeto e valor primitivos no Java podem ser representados como uma `String`. Quando um objeto é concatenado com uma `String`, ele é convertido em uma `String` e então as duas `Strings` são concatenadas.
- Se um `boolean` for concatenado com uma `String`, a palavra "`true`" ou "`false`" é utilizada para representar o valor `boolean`.
- Todos os objetos em Java têm um método especial chamado `toString` que retorna uma representação `String` do conteúdo do objeto. Quando um objeto é concatenado com uma `String`, a JVM chama implicitamente o método `toString` do objeto a fim de obter a representação string do objeto.
- Pode-se dividir grandes literais `String` em várias `Strings` menores e colocá-las em múltiplas linhas de código para melhorar a legibilidade, depois remontar as `Strings` utilizando concatenação.

Seção 6.5 Notas sobre a declaração e utilização de métodos

- Há três maneiras de chamar um método — utilizar o próprio nome de um método para chamar um outro método da mesma classe; utilizar 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; e utilizar o nome da classe e um ponto (`.`) para chamar um método `static` de uma classe.
- Há três maneiras de retornar o controle a uma instrução que chama um método. Se o método não retornar um resultado, o controle retornará quando o fluxo do programa alcançar a chave direita de fechamento do método ou quando a instrução

  ```
  return;
  ```

  for executada. Se o método retornar um resultado, a instrução

  ```
  return expressão;
  ```

  avalia a *expressão* e então imediatamente retorna o valor resultante ao chamador.

Seção 6.6 Pilhas de chamadas de método e quadros de pilha

- As pilhas são conhecidas como estruturas de dados do tipo último a entrar, primeiro a sair (*last-in, first-out* —– LIFO) — o último item inserido na pilha é o primeiro item que é removido dela.
- Um método chamado deve saber como retornar ao seu chamador, portanto o endereço de retorno do método de chamada é colocado na pilha de chamadas de método quando o método for chamado. Se uma série de chamadas de método ocorrer, os sucessivos endereços de retorno são empilhados na ordem "último a entrar, primeiro a sair", de modo que o último método a executar será o primeiro a retornar ao seu chamador.
- A pilha de chamadas de método contém a memória para as variáveis locais utilizadas em cada invocação de um método durante a execução de um programa. Esses dados são conhecidos como registro de ativação ou quadro de pilha da chamada de método. Quando uma chamada de método é feita, o quadro de pilha para ela é colocado na pilha de chamadas de método. Quando o método retorna ao seu chamador, a sua chamada do registro de ativação é retirada da pilha e as variáveis locais não são mais conhecidas para o programa.
- Se mais chamadas de método forem feitas do que o quadro de pilha pode armazenar na pilha de chamadas de método, ocorre um erro conhecido como estouro de pilha. O aplicativo compilará corretamente, mas sua execução causa um estouro de pilha.

Seção 6.7 Promoção e coerção de argumentos

- A promoção de argumentos converte o valor de um argumento para o tipo que o método espera receber no parâmetro correspondente.
- Regras de promoção se aplicam a expressões que contêm 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. Em casos em que as informações podem ser perdidas por causa da conversão, o compilador Java exige que se utilize um operador de coerção para forçar explicitamente que a conversão ocorra.

Seção 6.9 Estudo de caso: geração segura de números aleatórios

- Objetos da classe `SecureRandom` (pacote `java.security`) podem produzir valores aleatórios não determinísticos.
- O método `nextInt` `SecureRandom` gera um valor aleatório.

- A classe `SecureRandom` fornece uma outra versão do método `nextInt` que recebe um argumento `int` e retorna um valor a partir de 0, mas sem incluí-lo, até o valor do argumento.
- Números aleatórios em um intervalo podem ser gerados com

```
int 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.
- Os números aleatórios podem ser escolhidos a partir de intervalos de inteiro não consecutivos, como em

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

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

Seção 6.10 Estudo de caso: um jogo de azar; apresentando tipos `enum`

- Um tipo `enum` é introduzido pela palavra-chave `enum` e um nome de tipo. Como com qualquer classe, as chaves (`{` e `}`) delimitam o corpo de uma declaração `enum`. Entre as chaves há uma lista de constantes `enum`, cada uma representando um valor único separado por vírgula. Os identificadores em uma `enum` devem ser únicos. Pode-se atribuir variáveis de um tipo `enum` somente a constantes do tipo `enum`.
- Constantes também podem ser declaradas como variáveis `private static final`. Essas constantes, por convenção, são declaradas com todas as letras maiúsculas fazendo com que elas se destaquem no programa.

Seção 6.11 Escopo das declarações

- O escopo é a parte do programa em que uma entidade, como uma variável ou um método, pode ser referida pelo seu nome. Diz-se que essa entidade está "no escopo" para essa parte do programa.
- 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.
- O escopo de um método ou campo de uma classe é o corpo inteiro da classe. Isso permite que os métodos da classe utilizem nomes simples para chamar os outros métodos da classe e acessem os campos 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, este permanece sombreado até que o bloco termine a execução.

Seção 6.12 Sobrecarga de método

- O Java permite métodos sobrecarregados em uma classe, desde que os métodos tenham diferentes conjuntos de parâmetros (determinados pelo número, ordem e tipo de parâmetros).
- Métodos sobrecarregados são distinguidos por suas assinaturas — combinações dos nomes e número, tipos e ordem dos parâmetros dos métodos, mas não pelos tipos de retorno.

Exercícios de revisão

6.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um método é invocado com um(a) _____.

b) Uma variável conhecida somente dentro do método em que é declarada chama-se _____.

c) A instrução _____ em um método chamado pode ser utilizada para passar o valor de uma expressão de volta para o método de chamada.

d) A palavra-chave _____ indica que um método não retorna um valor.

e) Os dados podem ser adicionados ou removidos somente do(a) _____ de uma pilha.

f) As pilhas são conhecidas como estruturas de dados _____; o último item colocado (inserido) na pilha é o primeiro item retirado (removido) da pilha.

g) As três maneiras de retornar o controle de um método chamado a um chamador são _____, _____ e _____.

h) Um objeto da classe _____ produz números verdadeiramente aleatórios.

i) A pilha de execução de programas contém a memória criada para variáveis locais a cada invocação de método durante a execução de um programa. Esses dados, armazenados como parte da pilha de chamadas de método, são conhecidos como _____ ou _____ da chamada de método.

j) Se houver mais chamadas de método do que pode ser armazenado na pilha de execução do programa, um erro conhecido como _____ ocorrerá.

k) O _____ de uma declaração é a parte de um programa que pode referenciar a entidade na declaração pelo nome.

l) É possível ter diversos métodos com o mesmo nome que operam, separadamente, sobre diferentes tipos ou números de argumentos. Esse recurso é chamado de _____.

6.2 Para a classe `Craps` na Figura 6.8, declare o escopo de cada uma das seguintes entidades:

a) a variável `randomNumbers`.

b) a variável `die1`.

c) o método `rollDice`.

d) o método `main`.

e) a variável `sumOfDice`.

6.3 Escreva um aplicativo que teste se os exemplos de chamadas de método da classe `Math` mostrada na Figura 6.2 realmente produzem os resultados indicados.

6.4 Forneça o cabeçalho de método para cada um dos seguintes métodos.

a) O método `hypotenuse`, que aceita dois argumentos de ponto flutuante de precisão dupla `side1` e `side2` e retorna um resultado de ponto flutuante de dupla precisão.

b) O método `smallest`, que recebe três inteiros `x`, `y` e `z` e retorna um inteiro.

c) O método `instructions`, que não aceita nenhum argumento e não retorna um valor. [*Observação:* esses métodos são comumente utilizados para exibição de instruções para o usuário.]

d) O método `intToFloat`, que recebe um argumento `number` do tipo inteiro e retorna um `float`.

6.5 Encontre o erro em cada um dos seguintes segmentos de programa. Explique como corrigir o erro.

a)
```
void g()
{
       System.out.println("Inside method g");

       void h()
       {
          System.out.println("Inside method h");
       }
}
```

b)
```
int sum(int x, int y)
{
       int result;
       result = x + y;
}
```

c)
```
void f(float a);
{
       float a;
       System.out.println(a);
}
```

d)
```
void product()
{
       int a = 6, b = 5, c = 4, result;
       result = a * b * c;
       System.out.printf("Result is %d%n", result);
       return result;
}
```

6.6 Declare o método `sphereVolume` para calcular e retornar o volume da esfera. Utilize a seguinte instrução para calcular o volume:

```
double volume = (4.0 / 3.0) * Math.PI * Math.pow(radius, 3)
```

Escreva um aplicativo Java que solicita ao usuário o raio do tipo `double` de uma esfera, chama `sphereVolume` para calcular o volume e exibe o resultado.

Respostas dos exercícios de revisão

6.1 a) chamada de método. b) variável local. c) `return`. d) `void`. e) parte superior. f) último a entrar, primeiro a sair (LIFO). g) `return;` ou `return` *expressão*`;` ou encontre a chave direita de fechamento de um método. h) `SecureRandom`. i) registro de ativação, quadro de pilha. j) estouro de pilha. k) escopo. l) sobrecarga de método.

6.2 a) corpo de classe. b) bloco que define o corpo do método `rollDice`. c) corpo de classe. d) corpo de classe. e) bloco que define o corpo do método `main`.

6.3 A seguinte solução demonstra os métodos da classe `Math` na Figura 6.2:

```
1  // Exercício 6.3: MathTest.java
2  // Testando os métodos da classe Math
3  public class MathTest
4  {
5     public static void main(String[] args)
6     {
7           System.out.printf("Math.abs(23.7) = %f%n", Math.abs(23.7));
8        System.out.printf("Math.abs(0.0) = %f%n", Math.abs(0.0));
9        System.out.printf("Math.abs(-23.7) = %f%n", Math.abs(-23.7));
10       System.out.printf("Math.ceil(9.2) = %f%n", Math.ceil(9.2));
11       System.out.printf("Math.ceil(-9.8) = %f%n", Math.ceil(-9.8));
12       System.out.printf("Math.cos(0.0) = %f%n", Math.cos(0.0));
13       System.out.printf("Math.exp(1.0) = %f%n", Math.exp(1.0));
14       System.out.printf("Math.exp(2.0) = %f%n", Math.exp(2.0));
15       System.out.printf("Math.floor(9.2) = %f%n", Math.floor(9.2));
16       System.out.printf("Math.floor(-9.8) = %f%n", Math.floor(-9.8));
17       System.out.printf("Math.log(Math.E) = %f%n", Math.log(Math.E));
18       System.out.printf("Math.log(Math.E * Math.E) = %f%n",
19          Math.log(Math.E * Math.E));
20       System.out.printf("Math.max(2.3, 12.7) = %f%n", Math.max(2.3, 12.7));
21       System.out.printf("Math.max(-2.3, -12.7) = %f%n",
22          Math.max(-2.3, -12.7));
23       System.out.printf("Math.min(2.3, 12.7) = %f%n", Math.min(2.3, 12.7));
24       System.out.printf("Math.min(-2.3, -12.7) = %f%n",
25          Math.min(-2.3, -12.7));
26       System.out.printf("Math.pow(2.0, 7.0) = %f%n", Math.pow(2.0, 7.0));
27       System.out.printf("Math.pow(9.0, 0.5) = %f%n", Math.pow(9.0, 0.5));
28       System.out.printf("Math.sin(0.0) = %f%n", Math.sin(0.0));
29       System.out.printf("Math.sqrt(900.0) = %f%n", Math.sqrt(900.0));
30       System.out.printf("Math.tan(0.0) = %f%n", Math.tan(0.0));
31    } // fim de main
32 } // fim da classe MathTest
```

```
Math.abs(23.7) = 23.700000
Math.abs(0.0) = 0.000000
Math.abs(-23.7) = 23.700000
Math.ceil(9.2) = 10.000000
Math.ceil(-9.8) = -9.000000
Math.cos(0.0) = 1.000000
Math.exp(1.0) = 2.718282
Math.exp(2.0) = 7.389056
Math.floor(9.2) = 9.000000
Math.floor(-9.8) = -10.000000
Math.log(Math.E) = 1.000000
Math.log(Math.E * Math.E) = 2.000000
Math.max(2.3, 12.7) = 12.700000
Math.max(-2.3, -12.7) = -2.300000
Math.min(2.3, 12.7) = 2.300000
Math.min(-2.3, -12.7) = -12.700000
Math.pow(2.0, 7.0) = 128.000000
Math.pow(9.0, 0.5) = 3.000000
Math.sin(0.0) = 0.000000
Math.sqrt(900.0) = 30.000000
Math.tan(0.0) = 0.000000
```

6.4 a) `double hypotenuse(double side1, double side2)`

b) `int smallest(int x, int y, int z)`

c) `void instructions()`

d) `float intToFloat(int number)`

6.5 a) Erro: o método `h` é declarado dentro do método `g`.
Correção: mova a declaração de `h` para fora da declaração de `g`.

b) Erro: o método supostamente deve retornar um inteiro, mas não o faz.
Correção: exclua a variável `result` e coloque a instrução

```
return x + y;
```

no método ou adicione a seguinte instrução no fim do corpo de método:

```
return result;
```

c) Erro: ponto e vírgula após o parêntese direito da lista de parâmetros está incorreto e o parâmetro a não deve ser redeclarado no método.
Correção: exclua o ponto e vírgula após o parêntese direito da lista de parâmetros e exclua a declaração `float a;`.

d) Erro: o método retorna um valor quando supostamente não deveria.
Correção: altere o tipo de retorno de `void` para `int`.

6.6 A solução a seguir calcula o volume de uma esfera, utilizando o raio inserido pelo usuário:

```
1  // Exercício 6.6: Sphere.java
2  // Calcula o volume de uma esfera.
3  import java.util.Scanner;
4
5  public class Sphere
6  {
7    // obtém o raio a partir do usuário e exibe o volume da esfera
8    public static void main(String[] args)
9    {
10     Scanner input = new Scanner(System.in);
11
12     System.out.print("Enter radius of sphere: ");
13     double radius = input.nextDouble();
14
15     System.out.printf("Volume is %f%n", sphereVolume(radius));
16   } // fim do método determineSphereVolume
17
18   // calcula e retorna volume de esfera
19   public static double sphereVolume(double radius)
20   {
21     double volume = (4.0 / 3.0) * Math.PI * Math.pow(radius, 3);
22     return volume;
23   } // fim do método sphereVolume
24 } // fim da classe Sphere
```

```
Enter radius of sphere: 4
Volume is 268.082573
```

Questões

6.7 Qual é o valor de x depois que cada uma das seguintes instruções é executada?

a) `x = Math.abs(7.5);`

b) `x = Math.floor(7.5);`

c) `x = Math.abs(0.0);`

d) `x = Math.ceil(0.0);`

e) `x = Math.abs(-6.4);`

f) `x = Math.ceil(-6.4);`

g) `x = Math.ceil(-Math.abs(-8 + Math.floor(-5.5)));`

6.8 (***Taxas de estacionamento***) Um estacionamento cobra uma tarifa mínima de R$ 2,00 para estacionar por até três horas. Um adicional de R$ 0,50 por hora *não necessariamente inteira* é cobrado após as três primeiras horas. A tarifa máxima para qualquer dado período de 24 horas é R$ 10,00. Suponha que nenhum carro fique estacionado por mais de 24 horas por vez. Escreva um aplicativo que calcule e exiba as tarifas de estacionamento para cada cliente que estacionou nessa garagem ontem. Você deve inserir as horas de estacionamento para cada cliente. O programa deve exibir a cobrança para o cliente atual e calcular e exibir o total dos recibos de ontem. Ele deve utilizar o método `calculateCharges` para determinar a tarifa para cada cliente.

6.9 (***Arredondando números***) `Math.floor` pode ser utilizado para arredondar valores ao número inteiro mais próximo — por exemplo,

```
y = Math.floor(x + 0.5);
```

arredondará o número x para o inteiro mais próximo e atribuirá o resultado a y. Escreva um aplicativo que lê valores `double` e utiliza a instrução anterior para arredondar cada um dos números para o inteiro mais próximo. Para cada número processado, exiba ambos os números, o original e o arredondado.

6.10 (***Arredondando números***) Para arredondar números em casas decimais específicas, utilize uma instrução como

```
y = Math.floor(x * 10 + 0.5) / 10;
```

que arredonda x para a casa decimal (isto é, a primeira posição à direita do ponto de fração decimal), ou

```
y = Math.floor(x * 100 + 0.5) / 100;
```

que arredonda x para a casa centesimal (isto é, a segunda posição à direita do ponto de fração decimal). Escreva um aplicativo que defina quatro métodos para arredondar um número x de várias maneiras:

a) `roundToInteger(number)`

b) `roundToTenths(number)`

c) `roundToHundredths(number)`

d) `roundToThousandths(number)`

Para cada leitura de valor, seu programa deve exibir o valor original, o número arredondado para o inteiro mais próximo, o número arredondado para o décimo mais próximo, o número arredondado para o centésimo mais próximo e o número arredondado para o milésimo mais próximo.

6.11 Responda cada uma das seguintes perguntas:

a) O que significa escolher números "aleatoriamente"?

b) Por que o método `nextInt` da classe `SecureRandom` é útil para simular jogos de azar?

c) Por que frequentemente é necessário escalonar ou deslocar os valores produzidos por um objeto `SecureRandom`?

d) Por que a simulação computadorizada de situações do mundo real é uma técnica útil?

6.12 Escreva instruções que atribuem inteiros aleatórios à variável n nos seguintes intervalos:

a) $1 \leq n \leq 2$.

b) $1 \leq n \leq 100$.

c) $0 \leq n \leq 9$.

d) $1000 \leq n \leq 1112$.

e) $-1 \leq n \leq 1$.

f) $-3 \leq n \leq 11$.

6.13 Escreva instruções que exibirão um número aleatório de cada um dos seguintes conjuntos:

a) 2, 4, 6, 8, 10.

b) 3, 5, 7, 9, 11.

c) 6, 10, 14, 18, 22.

6.14 (***Exponenciação***) Escreva um método `integerPower(base, exponent)` que retorne o valor de

$base^{exponente}$

Por exemplo, `integerPower(3, 4)` calcula 3^4 (ou `3 * 3 * 3 * 3`). Suponha que `exponent` seja um inteiro não zero, positivo, e `base`, um inteiro. Use uma instrução `for` ou `while` para controlar o cálculo. Não utilize métodos da classe `Math`. Incorpore esse método a um aplicativo que lê os valores inteiros para `base` e `exponent` e realiza o cálculo com o método `integerPower`.

6.15 (***Cálculos de hipotenusa***) Defina um método `hypotenuse` que calcula a hipotenusa de um triângulo retângulo quando são dados os comprimentos dos outros dois lados. O método deve tomar dois argumentos do tipo `double` e retornar a hipotenusa como um `double`. Incorpore esse método a um aplicativo que lê valores para `side1` e `side2` e realiza o cálculo com o método `hypotenuse`. Utilize os métodos `Math` `pow` e `sqrt` para determinar o tamanho da hipotenusa de cada um dos triângulos na Figura 6.15. [*Observação:* a classe `Math` também fornece o método `hypot` para realizar esse cálculo.]

Triângulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

Figura 6.15 | Valores para os lados dos triângulos na Questão 6.15.

6.16 (***Múltiplos***) Escreva um método `isMultiple` que determina um par de inteiros se o segundo inteiro for um múltiplo do primeiro. O método deve aceitar dois argumentos inteiros e retornar `true` se o segundo for um múltiplo do primeiro e `false` caso contrário. [*Dica:* utilize o operador de módulo.] Incorpore esse método a um aplicativo que insere uma série de pares inteiros (um par por vez) e determina se o segundo valor em cada par é um múltiplo do primeiro.

6.17 (***Par ou ímpar***) Escreva um método `isEven` que utiliza o operador de resto (%) para determinar se um inteiro é par. O método deve levar um argumento inteiro e retornar `true` se o número inteiro for par, e `false`, caso contrário. Incorpore esse método a um aplicativo que insere uma sequência de inteiros (um por vez) e determina se cada um é par ou ímpar.

6.18 (***Exibindo um quadrado de asteriscos***) Escreva um método `squareOfAsterisks` que exibe um quadrado sólido (o mesmo número de linhas e colunas) de asteriscos cujo lado é especificado no parâmetro inteiro `side`. Por exemplo, se `side` for `4`, o método deverá exibir

```
****
****
****
****
```

Incorpore esse método a um aplicativo que lê um valor inteiro para `side` a partir da entrada fornecida pelo usuário e gera saída dos asteriscos com o método `squareOfAsterisks`.

6.19 (***Exibindo um quadrado de qualquer caractere***) Modifique o método criado no Exercício 6.18 para receber um segundo parâmetro do tipo `char` chamado `fillCharacter`. Forme o quadrado utilizando o `char` fornecido como um argumento. Portanto, se `side` for `5` e `fillCharacter` for `#`, o método deve exibir

```
#####
#####
#####
#####
#####
```

Utilize a seguinte instrução (em que `input` é um objeto `Scanner`) para ler um caractere do usuário no teclado:

```
char fill = input.next().charAt(0);
```

6.20 (***Área de círculo***) Escreva um aplicativo que solicita ao usuário o raio de um círculo e utiliza um método chamado `circleArea` para calcular a área do círculo.

6.21 (***Separando dígitos***) Escreva métodos que realizam cada uma das seguintes tarefas:

a) Calcule a parte inteiro do quociente quando o inteiro `a` é dividido pelo inteiro `b`.

b) Calcule o resto inteiro quando o inteiro `a` é dividido por inteiro `b`.

c) Utilize métodos desenvolvidos nas partes (a) e (b) para escrever um método `displayDigits` que recebe um inteiro entre `1` e `99999` e o exibe como uma sequência de dígitos, separando cada par de dígitos por dois espaços. Por exemplo, o inteiro `4562` deve aparecer como

```
4  5  6  2
```

Incorpore os métodos em um aplicativo que insere um número inteiro e chama `displayDigits` passando para o método o número inteiro inserido. Exiba os resultados.

6.22 (***Conversões de temperatura***) Implemente os seguintes métodos inteiros:

a) O método `celsius` retorna o equivalente em Celsius de uma temperatura em Fahrenheit utilizando o cálculo

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

b) O método `fahrenheit` retorna o equivalente em Fahrenheit de uma temperatura em Celsius utilizando o cálculo

```
fahrenheit = 9.0 / 5.0 * celsius + 32;
```

c) Utilize os métodos nas partes (a) e (b) para escrever um aplicativo que permite ao usuário inserir uma temperatura em Fahrenheit e exibir o equivalente em Celsius ou inserir uma temperatura em Celsius e exibir o equivalente em Fahrenheit.

6.23 (***Localize o mínimo***) Escreva um método `minimum3` que retorna o menor dos três números de ponto flutuante. Utilize o método `Math.min` para implementar `minimum3`. Incorpore o método a um aplicativo que lê três valores do usuário, determina o menor valor e exibe o resultado.

6.24 (***Números perfeitos***) Dizemos que um número inteiro é um *número perfeito* se a soma de seus fatores, incluindo 1 (mas não o próprio número), for igual ao número. Por exemplo, 6 é um número perfeito porque 6 = 1 + 2 + 3. Escreva um método `isPerfect` que determina se parâmetro `number` é um número perfeito. Utilize esse método em um applet que determina e exibe todos os números perfeitos entre 1 e 1.000. Exiba os fatores de cada número perfeito confirmando que ele é de fato perfeito. Desafie o poder de computação do seu computador testando números bem maiores que 1.000. Exiba os resultados.

6.25 (***Números primos***) Um número inteiro positivo é *primo* se for divisível apenas por 1 e por ele mesmo. Por exemplo, 2, 3, 5 e 7 são primos, mas 4, 6, 8 e 9 não são. O número 1, por definição, não é primo.

a) Escreva um método que determina se um número é primo.

b) Utilize esse método em um aplicativo que determina e exibe todos os números primos menores que 10.000. Quantos números até 10.000 você precisa testar a fim de assegurar que encontrou todos os primos?

c) Inicialmente, você poderia pensar que $n/2$ é o limite superior que deve ser testado para ver se um número é primo, mas você precisa ir apenas até a raiz quadrada de n. Reescreva o programa e execute-o de ambas as maneiras.

6.26 **(*Invertendo dígitos*)** Escreva um método que recebe um valor inteiro e retorna o número com seus dígitos invertidos. Por exemplo, dado o número 7.631, o método deve retornar 1.367. Incorpore o método a um aplicativo que lê um valor a partir da entrada fornecida pelo usuário e exibe o resultado.

6.27 **(*Máximo divisor comum*)** O *máximo divisor comum (MDC)* de dois inteiros é o maior inteiro que é divisível por cada um dos dois números. Escreva um método `mdc` que retorna o máximo divisor comum de dois inteiros. [*Dica:* você poderia querer utilizar o algoritmo de Euclides. Você pode encontrar informações sobre isso em `en.wikipedia.org/wiki/Euclidean_algorithm`.] Incorpore o método a um aplicativo que lê dois valores do usuário e exibe o resultado.

6.28 Escreva um método `qualityPoints` que insere a média de um aluno e retorna `4` se for 90 a 100, `3` se 80 a 89, `2` se 70 a 79, `1` se 60 a 69 e `0` se menor que 60. Incorpore o método a um aplicativo que lê um valor a partir do usuário e exibe o resultado.

6.29 **(*Cara ou coroa*)** Escreva um aplicativo que simula o jogo de cara ou coroa. Deixe o programa lançar uma moeda toda vez que o usuário escolher a opção "`Toss Coin`" no menu. Conte o número de vezes que cada lado da moeda aparece. Exiba os resultados. O programa deve chamar um método `flip` separado que não aceita argumentos e retorna um valor a partir de um `Coin enum` (`HEADS` e `TAILS`). [*Observação*: se o programa simular de modo realista o arremesso de moeda, cada lado da moeda deve aparecer aproximadamente metade das vezes.]

6.30 **(*Adivinhe o número*)** Escreva um aplicativo que execute "adivinhe o número" como mostrado a seguir: seu programa escolhe o número a ser adivinhado selecionando um inteiro aleatório no intervalo de 1 a 1.000. O aplicativo exibe o prompt `Guess a number between 1 and 1000 [adivinhe um número entre 1 e 1000]`. O jogador insere uma primeira suposição. Se o palpite do jogador estiver incorreto, seu programa deve exibir `Too high. Try again [Muito alto. Tente novamente]` ou `Too low. Try again [Muito baixo. Tente novamente]` para ajudar o jogador a alcançar a resposta correta. O programa deve solicitar ao usuário o próximo palpite. Quando o usuário insere a resposta correta, exibe `Congratulations. You guessed the number. [Parabéns, você adivinhou o número!]` e permite que o usuário escolha se quer jogar novamente. [*Observação*: a técnica de adivinhação empregada nesse problema é semelhante a uma pesquisa binária, discutida no Capítulo 19, "Pesquisa, classificação e Big O".]

6.31 **(*Adivinhe a modificação de número*)** Modifique o programa do Exercício 6.30 para contar o número de adivinhações que o jogador faz. Se o número for 10 ou menos, exibe `Either you know the secret or you got lucky! [Você sabe o segredo ou tem muita sorte!]`; se o jogador adivinhar o número em 10 tentativas, exiba `Aha! You know the secret! [Aha! Você sabe o segredo!]`; se o jogador fizer mais que 10 adivinhações, exiba `You should be able to do better! [Você deve ser capaz de fazer melhor]`. Por que esse jogo não deve precisar de mais que 10 suposições? Bem, com cada "boa adivinhação" o jogador deve ser capaz de eliminar a metade dos números, depois a metade dos números restantes, e assim por diante.

6.32 **(*Distância entre pontos*)** Escreva um método `distance` para calcular a distância entre dois pontos $(x1, y1)$ e $(x2, y2)$. Todos os números e valores de retorno devem ser do tipo `double`. Incorpore esse método a um aplicativo que permite que o usuário insira as coordenadas de pontos.

6.33 **(*Modificação do jogo Craps*)** Modifique o programa de jogo de dados craps da Figura 6.8 para permitir apostas. Inicialize a variável `bankBalance` como 1.000 dólares. Peça ao jogador que insira um `wager`. Verifique se `wager` é menor ou igual a `bankBalance` e, se não for, faça o usuário reinserir `wager` até um `wager` válido ser inserido. Então, execute um jogo de dados. Se o jogador ganhar, aumente `bankBalance` por `wager` e exiba o novo `bankBalance`. Se o jogador perder, diminua `bankBalance` por `wager`, exiba o novo `bankBalance`, verifique se `bankBalance` tornou-se zero e, se isso tiver ocorrido, exiba a mensagem `"Sorry. You busted!" ["Desculpe, mas você faliu!"]`. À medida que o jogo se desenvolve, exiba várias mensagens para criar uma "conversa", como `"Oh, you're going for broke, huh?" ["Oh, parece que você vai quebrar, hein?"]` ou `"Aw c'mon, take a chance!" ["Ah, vamos lá, dê uma chance para sua sorte"]` ou `"You're up big. Now's the time to cash in your chips!" [Você está montado na grana. Agora é hora de trocar essas fichas e embolsar o dinheiro!"]`. Implemente a "conversa" como um método separado que escolhe aleatoriamente a string a ser exibida.

6.34 **(*Tabela de números binários, octais e hexadecimais*)** Escreva um aplicativo que exibe uma tabela de equivalentes binários, octais e hexadecimais dos números decimais no intervalo de 1 a 256. Se você não estiver familiarizado com esses sistemas de números, leia primeiro o Apêndice J, em inglês, na Sala Virtual do Livro.

Fazendo a diferença

À medida que o preço dos computadores cai, torna-se viável para cada estudante, apesar da circunstância econômica, ter um computador e utilizá-lo na escola. Isso cria grandes oportunidades para aprimorar a experiência educativa de todos os estudantes em todo o mundo, conforme sugerido pelos cinco exercícios a seguir. [*Observação*: verifique iniciativas como One Laptop Per Child Project (`www.laptop.org`). Pesquise também laptops "verdes" — quais são as principais características amigáveis ao meio ambiente desses dispositivos? Consulte a Electronic Product Environmental Assessment Tool (`www.epeat.net`), que pode ajudar a avaliar o grau de responsabilidade ambiental "greenness" de computadores desktop, notebooks e monitores para ajudar a decidir que produtos comprar.]

6.35 **(*Instrução assistida por computador*)** O uso de computadores na educação é chamado *instrução assistida por computador* (CAI). Escreva um programa que ajudará um aluno da escola elementar a aprender multiplicação. Utilize um objeto `SecureRandom` para produzir dois inteiros positivos de um algarismo. O programa deve então fazer ao usuário uma pergunta, como

```
Quanto é 6 vezes 7?
```

O aluno insere então a resposta. Em seguida, o programa verifica a resposta do aluno. Se estiver correta, exiba a mensagem `"Muito bem!"` e faça uma outra pergunta de multiplicação. Se a resposta estiver errada, exiba a mensagem `"Não. Por favor, tente de novo."` e deixe que o aluno tente a mesma pergunta várias vezes até que por fim ele acerte. Um método separado deve ser utilizado para gerar cada nova pergunta. Esse método deve ser chamado uma vez quando a aplicação inicia a execução e toda vez que o usuário responde a pergunta corretamente.

6.36 **(*Instrução auxiliada por computador: reduzindo a fadiga do aluno*)** Um problema em ambientes CAI é a fadiga do aluno. Isso pode ser reduzido variando-se as respostas do computador para prender a atenção do aluno. Modifique o programa da Questão 6.35 para que vários comentários sejam exibidos para cada resposta como mostrado a seguir:

Possibilidades para uma resposta correta:

```
Muito bom!
Excelente!
Bom trabalho!
Mantenha um bom trabalho!
```

Possibilidades para uma resposta incorreta:

```
Não. Por favor, tente de novo.
Errado. Tente mais uma vez.
Não desista!
Não. Continue tentando.
```

Utilize a geração de números aleatórios para escolher um número de 1 a 4 que será utilizado para selecionar uma de quatro respostas adequadas a cada resposta correta ou incorreta. Utilize uma instrução `switch` para emitir as respostas.

6.37 **(*Instrução auxiliada por computador: monitorando o desempenho do aluno*)** Sistemas mais sofisticados de instruções auxiliadas por computador monitoram o desempenho do aluno durante um período de tempo. A decisão sobre um novo tópico frequentemente é baseada no sucesso do aluno com tópicos prévios. Modifique o programa de Exercício 6.36 para contar o número de respostas corretas e incorretas digitadas pelo aluno. Depois que o aluno digitar 10 respostas, seu programa deve calcular a porcentagem das que estão corretas. Se a porcentagem for menor que 75%, exiba `"Peça ajuda extra ao seu professor."` e, então, reinicialize o programa para que outro estudante possa tentá-lo. Se a porcentagem for 75% ou maior, exiba `"Parabéns, você está pronto para avançar para o próximo nível!"` e, então, reinicialize o programa para que outro estudante possa tentá-lo.

6.38 **(*Instrução auxiliada por computador: níveis de dificuldade*)** As questões 6.35 a 6.37 desenvolveram um programa de instrução assistida por computador a fim de ajudar a ensinar multiplicação para um aluno do ensino fundamental. Modifique o programa para permitir que o usuário insira um nível de dificuldade. Em um nível de dificuldade 1, o programa deve utilizar apenas números de um único dígito nos problemas; em um nível de dificuldade 2, os números com dois dígitos, e assim por diante.

6.39 **(*Instrução auxiliada por computador: variando os tipos de problema*)** Modifique o programa da Questão 6.38 a fim de permitir ao usuário selecionar um tipo de problema de aritmética a ser estudado. Uma opção de `1` significa apenas problemas de adição, `2` significa apenas problemas de subtração, `3`, de multiplicação, `4`, de divisão e `5`, uma combinação aleatória de problemas de todos esses tipos.