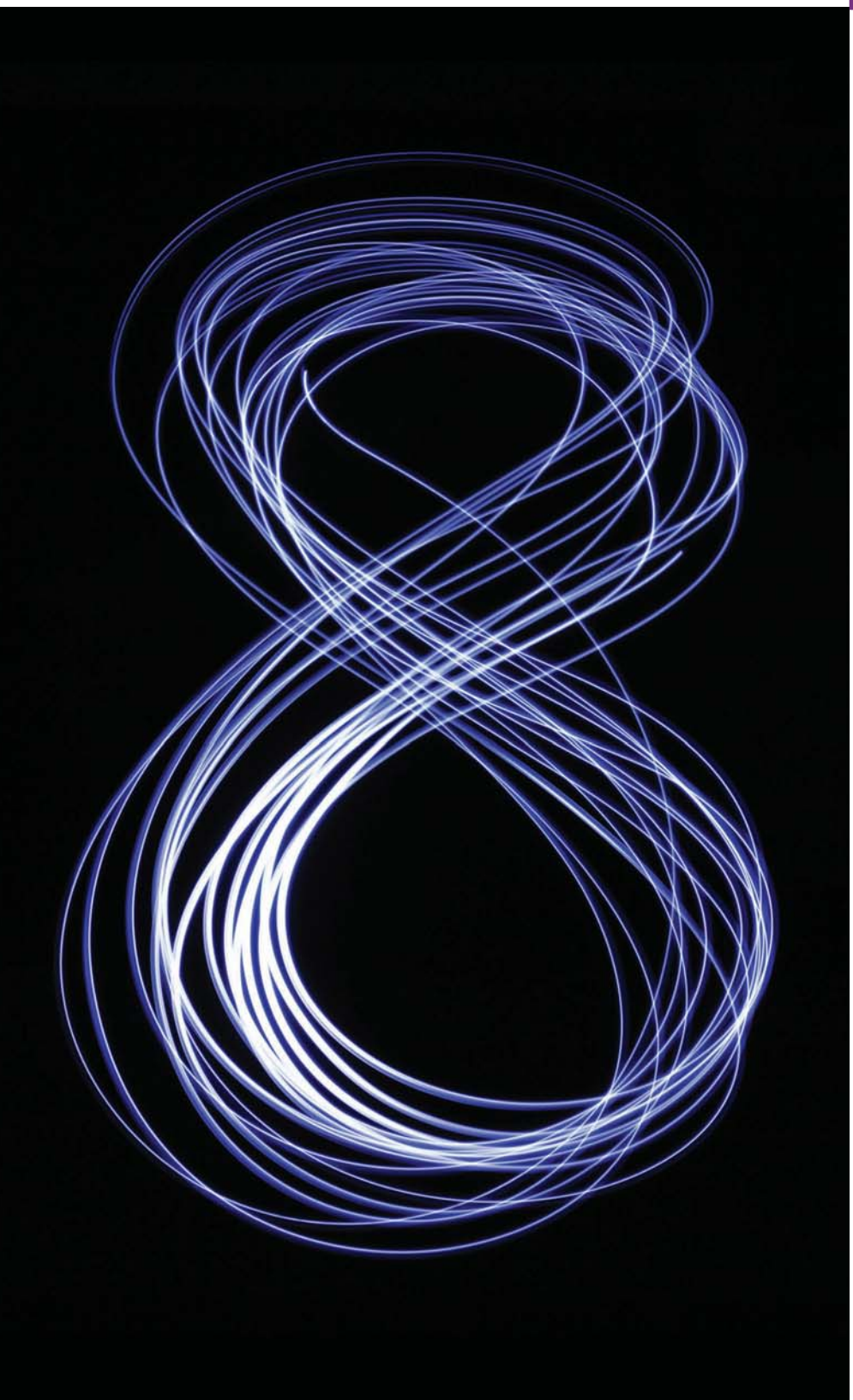
Classes e objetos: um exame mais profundo

8



Este é um mundo para esconder virtudes?
— William Shakespeare

Mas o que, para servir nossos propósitos particulares, proíbe trapacear nossos amigos?
— Charles Churchill

Mas, sobretudo, sê a ti próprio fiel.
— William Shakespeare

Objetivos

Neste capítulo, você irá:

- Usar a instrução `throw` para indicar que ocorreu um problema.
- Utilizar a palavra-chave `this` em um construtor para chamar outro construtor na mesma classe.
- Usar variáveis e métodos `static`.
- Importar membros `static` de uma classe.
- Usar o tipo `enum` para criar conjuntos de constantes com identificadores únicos.
- Declarar constantes `enum` com parâmetros.
- Utilizar `BigDecimal` para cálculos monetários precisos.

Sumário

8.1 Introdução

Agora faremos uma análise mais profunda da construção de classes, controle de acesso a membros de uma classe e criação de construtores. Mostraremos como usar `throw` para lançar uma exceção a fim de indicar que ocorreu um problema (a Seção 7.5 discutiu exceções `catch`) e a palavra-chave `this` para permitir que um construtor chame convenientemente outro construtor da mesma classe. Discutiremos a *composição* — uma capacidade que permite a uma classe conter referências a objetos de outras classes como membros. Reexaminaremos a utilização dos métodos *set* e *get*. Lembre-se de que a Seção 6.10 introduziu o tipo `enum` básico para declarar um conjunto de constantes, neste capítulo, discutiremos o relacionamento entre tipos e classes `enum`, demonstrando que um tipo `enum`, como ocorre com uma classe, pode ser declarado no seu próprio arquivo com construtores, métodos e campos. O capítulo também discute os membros da classe `static` e variáveis de instância `final` em detalhes. Mostramos uma relação especial entre as classes no mesmo pacote. Por fim, demonstramos como usar a classe `BigDecimal` para realizar cálculos monetários precisos. Dois outros tipos de classes — classes interiores anônimas e classes aninhadas — são discutidos no Capítulo 12.

8.2 Estudo de caso da classe `Time`

Nosso primeiro exemplo consiste em duas classes — `Time1` (Figura 8.1) e `Time1Test` (Figura 8.2). A classe `Time1` representa a hora do dia. O método `main` da classe `Time1Test` cria um objeto da classe `Time1` e chama seus métodos. A saída desse programa é mostrada na Figura 8.2.

Declaração da classe `Time1`

As variáveis de instância `private int` `hour`, `minute` e `second` da classe `Time1` (linhas 6 a 8) representam a hora no formato de data/hora universal (formato de relógio de 24 horas em que as horas estão no intervalo de 0 a 23, e minutos e segundos estão no intervalo 0 a 59). `Time1` contém os métodos `public` `setTime` (linhas 12 a 25), `toUniversalString` (linhas 28 a 31) e `toString` (linhas 34 a 39). Esses métodos também são chamados de **serviços `public`** ou **interface `public`** que a classe fornece para seus clientes.

```
1  // Figura 8.1: Time1.java
2  // Declaração de classe Time1 mantém a data/hora no formato de 24 horas.
3
4  public class Time1
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // configura um novo valor de tempo usando hora universal; lança uma
11    // exceção se a hora, minuto ou segundo for inválido
12    public void setTime(int hour, int minute, int second)
13    {
14       // valida hora, minuto e segundo
15       if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
16          second < 0 || second >= 60)
```

continua

continuação

```
17          {
18             throw new IllegalArgumentException(
19                "hour, minute and/or second was out of range");
20          }
21
22          this.hour = hour;
23          this.minute = minute;
24          this.second = second;
25       }
26
27       // converte em String no formato de data/hora universal (HH:MM:SS)
28       public String toUniversalString()
29       {
30          return String.format("%02d:%02d:%02d", hour, minute, second);
31       }
32
33       // converte em String no formato padrão de data/hora (H:MM:SS AM ou PM)
34       public String toString()
35       {
36          return String.format("%d:%02d:%02d %s",
37             ((hour == 0 || hour == 12) ? 12 : hour % 12),
38             minute, second, (hour < 12 ? "AM" : "PM"));
39       }
40    } // fim da classe Time1
```

Figura 8.1 | Declaração da classe `Time1` mantém a data/hora no formato de 24 horas.

Construtor padrão

Nesse exemplo, a classe `Time1` *não* declara um construtor, assim o compilador fornece um construtor padrão. Cada variável de instância recebe implicitamente o valor `int` padrão. As variáveis de instância também podem ser inicializadas quando declaradas no corpo da classe, utilizando-se a mesma sintaxe de inicialização de uma variável local.

Método `setTime` e lançamento de exceções

O método `setTime` (linhas 12 a 25) é um método `public` que declara três parâmetros `int` e os utiliza para configurar a data/hora. As linhas 15 e 16 testam cada argumento para determinar se o valor está fora do intervalo adequado. O valor `hour` deve ser maior ou igual a `0` e menor que `24`, porque o formato de data/hora universal representa as horas como números inteiros de 0 a 23 (por exemplo, 1 PM é 13 horas e 11 PM é 23 horas; meia-noite é 0 hora e meio-dia é 12 horas). Do mesmo modo, os valores `minute` e `second` devem ser maiores ou iguais a `0` e menor que `60`. Para valores fora desses intervalos, `setTime` **lança uma exceção** do tipo `IllegalArgumentException` (linhas 18 e 19), que notifica o código do cliente que um argumento inválido foi passado para o método. Como vimos na Seção 7.5, você pode usar `try...catch` para capturar exceções e tentar recuperar a partir delas, o que faremos na Figura 8.2. A expressão de criação de instância de classe na **instrução `throw`** (Figura 8.1, linha 18) cria um novo objeto do tipo `IllegalArgumentException`. Os parênteses após o nome da classe indicam uma chamada para o construtor `IllegalArgumentException`. Nesse caso, chamamos o construtor que permite especificar uma mensagem de erro personalizada. Depois que o objeto de exceção é criado, a instrução `throw` termina imediatamente o método `setTime` e a exceção é retornada para o método chamador que tentou definir a data/hora. Se todos os valores de argumento são válidos, as linhas 22 a 24 os atribuem às variáveis de instância `hour`, `minute` e `second`.

Observação de engenharia de software 8.1

Para um método como `setTime` na Figura 8.1, valide todos os argumentos do método antes de usá-los para definir os valores das variáveis de instância a fim de garantir que os dados do objeto só sejam modificados se todos os argumentos forem válidos.

Método `toUniversalString`

O método `toUniversalString` (linhas 28 a 31) não recebe nenhum argumento e retorna uma `String` no *formato de data/hora universal*, cada um consistindo em dois dígitos para hora, minuto e segundo — lembre-se de que você pode usar o flag `0` em uma especificação de formato `printf` (por exemplo, `"%02d"`) para exibir zeros à esquerda para um valor que não usa todas as posições de caracteres na largura especificada de campo. Por exemplo, se a data/hora fosse 1:30:07 PM, o método retornaria `13:30:07`. A linha 30 utiliza o método `static` **`format`** da classe `String` para retornar uma `String` que contém os valores `hour`, `minute` e

second formatados, cada um com dois dígitos e possivelmente um 0 à esquerda (especificado com o flag 0). O método format é semelhante ao método System.out.printf, exceto que format *retorna* uma String formatada em vez de exibi-la em uma janela de comando. A String formatada é retornada pelo método toUniversalString.

Método toString

O método toString (linhas 34 a 39) não recebe argumentos e retorna uma String em *formato de data/hora padrão*, que consiste nos valores hour, minute e second separados por dois-pontos e seguidos por AM ou PM (por exemplo, 11:30:17 AM ou 1:27:06 PM). Como o método toUniversalString, o método toString utiliza o método static String format para formatar os valores minute e second como valores de dois dígitos com zeros à esquerda, se necessário. A linha 37 utiliza um operador condicional (?:) para determinar o valor para hour na String — se hour for 0 ou 12 (AM ou PM), ele aparece como 12; caso contrário, ele aparece como um valor entre 1 e 11. O operador condicional na linha 30 determina se AM ou PM será retornado como parte da String.

Lembre-se de que todos os objetos em Java têm um método toString que retorna uma representação String do objeto. Escolhemos retornar uma String que contém a data/hora no formato de data/hora padrão. O método toString é chamado *implicitamente* sempre que um objeto Time1 aparece no código em que uma String é necessária, como o valor para gerar a saída com um especificador de formato %s em uma chamada para System.out.printf. Você também pode chamar toString para obter *explicitamente* uma representação de String de um objeto Time.

Utilizando a classe Time1

A classe Time1Test (Figura 8.2) usa a classe Time1. A linha 9 declara e cria um objeto Time1. O operador new invoca implicitamente o construtor padrão da classe Time1, porque Time1 não declara nenhum construtor. Para confirmar que o objeto Time1 foi inicializado adequadamente, a linha 12 chama o método displayTime private (linhas 35 a 39) que, por sua vez, chama os métodos toUniversalString e toString do objeto Time1 para gerar a data/hora no formato de data/hora universal e no formato de data/hora padrão, respectivamente. Note que toString poderia ter sido chamado implicitamente aqui em vez de explicitamente. Em seguida, a linha 16 invoca o método setTime do objeto time sem mudar a data/hora. Então, a linha 17 chama o displayTime novamente para gerar a data/hora em ambos os formatos a fim de confirmar que ela foi configurada corretamente.

Observação de engenharia de software 8.2

Lembre-se do que foi discutido no Capítulo 3, de que os métodos declarados com o modificador de acesso private só podem ser chamados por outros métodos da classe em que os métodos private são declarados. Esses métodos são comumente chamados de ***métodos utilitários*** *ou* ***métodos auxiliares*** *porque eles são tipicamente utilizados para suportar a operação dos outros métodos da classe.*

```
1  // Figura 8.2: Time1Test.java
2  // objeto Time1 utilizado em um aplicativo.
3
4  public class Time1Test
5  {
6     public static void main(String[] args)
7     {
8        // cria e inicializa um objeto Time1
9        Time1 time = new Time1(); // invoca o construtor Time1
10
11       // gera saída de representações de string da data/hora
12       displayTime("After time object is created", time);
13       System.out.println();
14
15       // altera a data/hora e gera saída da data/hora atualizada
16       time.setTime(13, 27, 6);
17       displayTime("After calling setTime", time);
18       System.out.println();
19
20       // tenta definir data/hora com valores inválidos
21       try
22       {
23          time.setTime(99, 99, 99); // todos os valores fora do intervalo
24       }
25       catch (IllegalArgumentException e)
26       {
27          System.out.printf("Exception: %s%n%n", e.getMessage());
28       }
```

continua

continuação

```
29
30          // exibe a data/hora após uma tentativa de definir valores inválidos
31          displayTime("After calling setTime with invalid values", time);
32       }
33
34       // exibe um objeto Time1 nos formatos de 24 horas e 12 horas
35       private static void displayTime(String header, Time1 t)
36       {
37          System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
38             header, t.toUniversalString(),t.toString());
39       }
40    } // fim da classe Time1Test
```

```
After time object is created
Universal time: 00:00:00
Standard time: 12:00:00 AM

After calling setTime
Universal time: 13:27:06
Standard time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After calling setTime with invalid values
Universal time: 13:27:06
Standard time: 1:27:06 PM
```

Figura 8.2 | Objeto `Time1` utilizado em um aplicativo.

Chamando o método `Time1` `setTime` com valores inválidos

Para ilustrar que o método `setTime` *valida* seus argumentos, a linha 23 chama o método `setTime` com argumentos *inválidos* de `99` para `hour`, `minute` e `second`. Essa instrução é colocada em um bloco `try` (linhas 21 a 24) se `setTime` lançar uma `IllegalArgumentException`, o que fará, uma vez que todos os argumentos são inválidos. Quando isso ocorre, a exceção é capturada nas linhas 25 a 28, e a linha 27 exibe a mensagem de erro da exceção chamando o método `getMessage`. A linha 31 gera a data/hora mais uma vez nos dois formatos para confirmar que `setTime` *não* alterou a data/hora quando argumentos inválidos forem fornecidos.

Engenharia de software da declaração da classe `Time1`

Considere as várias questões de projeto classe com relação à classe `Time1`. As variáveis de instância `hour`, `minute` e `second` são declaradas `private`. A representação real dos dados utilizada dentro da classe não diz respeito aos clientes da classe. Por exemplo, seria perfeitamente razoável para `Time1` representar a data/hora internamente como o número de segundos a partir da meia-noite ou o número de minutos e segundos a partir da meia-noite. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados sem estarem cientes disso. (O Exercício 8.5 pede para você representar a data/hora na classe `Time2` da Figura 8.5 como o número de segundos a partir da meia-noite e mostrar que de fato nenhuma alteração está visível aos clientes da classe.)

Observação de engenharia de software 8.3

Classes simplificam a programação porque o cliente só pode utilizar os métodos `public` de uma classe. Normalmente, esses métodos são direcionados aos clientes em vez de à implementação. Os clientes não estão cientes de, nem envolvidos em, uma implementação da classe. Eles geralmente se preocupam com o que *a classe faz, mas não* como *a classe faz isso.*

Observação de engenharia de software 8.4

As interfaces mudam com menos frequência que as implementações. Quando uma implementação muda, o código dependente de implementação deve alterar correspondentemente. Ocultar a implementação reduz a possibilidade de que outras partes do programa irão se tornar dependentes dos detalhes sobre a implementação da classe.

Java SE 8 — Date/Time API

O exemplo desta seção e os vários exemplos mais adiante neste capítulo demonstram vários conceitos de implementação de classe daquelas que representam datas e horas. No desenvolvimento Java profissional, em vez de construir suas próprias classes de

data e hora, você normalmente reutiliza aquelas fornecidas pela API Java. Embora o Java sempre tenha tido classes para manipular datas e horas, o Java SE 8 introduz uma nova **Date/Time API** — definida pelas classes no pacote `java.time` — aplicativos construídos com o Java SE 8 devem usar as capacidades da Date/Time API, em vez das versões anteriores do Java. A nova API corrige vários problemas com as classes mais antigas e fornece capacidades mais robustas, mais fáceis de usar para manipular datas, horas, fusos horários, calendários e mais. Usamos alguns dos recursos da Date/Time API no Capítulo 23. Você pode aprender mais sobre as classes da Date/Time API em:

```
download.java.net/jdk8/docs/api/
```

8.3 Controlando o acesso a membros

Os modificadores de acesso `public` e `private` controlam o acesso a variáveis e métodos de uma classe. No Capítulo 9, introduziremos o modificador de acesso adicional `protected`. O principal objetivo dos métodos `public` é apresentar aos clientes da classe uma visualização dos serviços que a classe fornece (isto é, a interface `public` dela). Os clientes não precisam se preocupar com a forma como a classe realiza suas tarefas. Por essa razão, as variáveis `private` e os métodos `private` da classe (isto é, seus *detalhes de implementação*) *não* são acessíveis aos clientes.

A Figura 8.3 demonstra que os membros da classe `private` *não* são acessíveis fora da classe. As linhas 9 a 11 tentam acessar diretamente as variáveis de instância `hour`, `minute` e `second` de `private` da `time` do objeto `Time1`. Quando esse programa é compilado, o compilador gera mensagens de erro de que esses membros `private` não são acessíveis. Esse programa supõe que a classe `Time1` da Figura 8.1 é utilizada.

```
1  // Figura 8.3: MemberAccessTest.java
2  // Membros privados da classe Time1 não são acessíveis.
3  public class MemberAccessTest
4  {
5     public static void main(String[] args)
6     {
7        Time1 time = new Time1(); // cria e inicializa o objeto Time1
8
9        time.hour = 7; // erro: hour tem acesso privado em Time1
10       time.minute = 15; // erro: minute tem acesso privado em Time1
11       time.second = 30; // erro: second tem acesso privado em Time1
12    }
13 } // fim da classe MemberAccessTest
```

```
MemberAccessTest.java:9: hour tem acesso privado em Time1
     time.hour = 7; // erro: hour tem acesso privado em Time1
        ^
MemberAccessTest.java:10: minute tem acesso privado em Time1
     time.minute = 15; // erro: minute tem acesso privado em Time1
        ^
MemberAccessTest.java:11: second tem acesso privado em Time1
     time.second = 30; // erro: second tem acesso privado em Time1
        ^
3 errors
```

Figura 8.3 | Membros privados da classe `Time1` não são acessíveis.

Erro comum de programação 8.1

Uma tentativa por um método que não é membro de uma classe de acessar um membro `private` dessa classe é um erro de compilação.

8.4 Referenciando membros do objeto atual com a referência `this`

Cada objeto pode acessar uma *referência a si próprio* com a palavra-chave `this` (às vezes chamada de **referência `this`**). Quando um método de instância é chamado para um objeto particular, o corpo do método utiliza *implicitamente* a palavra-chave `this` para referenciar as variáveis de instância do objeto e outros métodos. Isso permite que o código da classe saiba qual objeto deve ser manipulado. Como veremos na Figura 8.4, você também pode usar a palavra-chave `this` *explicitamente* no corpo do método de uma

instância. A Seção 8.5 mostra uma outra utilização interessante de palavra-chave `this`. A Seção 8.11 explica por que a palavra-chave `this` não pode ser utilizada em um método `static`.

Agora demonstraremos o uso implícito e explícito da referência `this` (Figura 8.4). Esse exemplo é o primeiro em que declaramos *duas* classes em um único arquivo — a classe `ThisTest` é declarada nas linhas 4 a 11 e a classe `SimpleTime`, nas linhas 14 a 47. Fizemos isso para demonstrar que, quando você compila um arquivo `.java` contendo mais de uma classe, o compilador produz um arquivo separado da classe com a extensão `.class` para cada classe compilada. Nesse caso, dois arquivos separados são produzidos — `SimpleTime.class` e `ThisTest.class`. Quando um arquivo de código-fonte (`.java`) contém múltiplas declarações de classe, o compilador insere ambos os arquivos de classe para essas classes no *mesmo* diretório. Observe também na Figura 8.4 que só a classe `ThisTest` é declarada `public`. Um arquivo de código-fonte pode conter somente *uma* classe `public` — caso contrário, um erro de compilação ocorre. *As classes não `public`* só podem ser utilizadas por outras classes no *mesmo pacote*. Assim, nesse exemplo, a classe `SimpleTime` só pode ser usada pela classe `ThisTest`.

```
1  // Figura 8.4: ThisTest.java
2  // this utilizado implícita e explicitamente para referência a membros de um objeto.
3
4  public class ThisTest
5  {
6     public static void main(String[] args)
7     {
8        SimpleTime time = new SimpleTime(15, 30, 19);
9        System.out.println(time.buildString());
10    }
11 } // fim da classe ThisTest
12
13 // classe SimpleTime demonstra a referência "this"
14 class SimpleTime
15 {
16    private int hour; // 0-23
17    private int minute; // 0-59
18    private int second; // 0-59
19
20    // se o construtor utilizar nomes de parâmetro idênticos a
21    // nomes de variáveis de instância a referência "this" será
22    // exigida para distinguir entre os nomes
23    public SimpleTime(int hour, int minute, int second)
24    {
25       this.hour = hour; // configura a hora do objeto "this"
26       this.minute = minute; // configura o minuto do objeto "this"
27       this.second = second; // configura o segundo do objeto "this"
28    }
29
30    // utilizam "this" explícito e implícito para chamar toUniversalString
31    public String buildString()
32    {
33       return String.format("%24s: %s%n%24s: %s",
34          "this.toUniversalString()",this.toUniversalString(),
35          "toUniversalString()",toUniversalString());
36    }
37
38    // converte em String no formato de data/hora universal (HH:MM:SS)
39    public String toUniversalString()
40    {
41       // "this" não é requerido aqui para acessar variáveis de instância,
42       // porque o método não tem variáveis locais com os mesmos
43       // nomes das variáveis de instância
44       return String.format("%02d:%02d:%02d",
45          this.hour, this.minute, this.second);
46    }
47 } // fim da classe SimpleTime
```

```
this.toUniversalString(): 15:30:19
    toUniversalString(): 15:30:19
```

Figura 8.4 | `this` utilizado implícita e explicitamente como uma referência a membros de um objeto.

A classe `SimpleTime` (linhas 14 a 47) declara três variáveis de instância `private` — `hour`, `minute` e `second` (linhas 16 a 18). O construtor da classe (linha 23 a 28) recebe três argumentos `int` para inicializar um objeto `SimpleTime`. Mais uma vez, utilizamos nomes de parâmetro para o construtor (linha 23) que são *idênticos* aos nomes das variáveis de instância da classe (linhas 16 a 18), assim usamos a referência `this` para nos referirmos às variáveis de instância nas linhas 25 a 27.

Dica de prevenção de erro 8.1

A maioria dos IDEs emitirá um alerta se você afirmar `x = x;` em vez de `this.x = x;`. A instrução `x = x;` é muitas vezes chamada no-op (no operation).

O método `buildString` (linhas 31 a 36) retorna uma `String` criada por uma instrução que utiliza a referência `this` explícita e implicitamente. A linha 34 usa-a *explicitamente* para chamar o método `toUniversalString`. A linha 35 utiliza-a *implicitamente* para chamar o mesmo método. Ambas as linhas realizam a mesma tarefa. Em geral, você não utilizará `this` explicitamente para referenciar outros métodos dentro do objeto atual. Além disso, a linha 45 no método `toUniversalString` utiliza explicitamente a referência `this` para acessar cada variável de instância. Isso *não* é necessário aqui, porque o método *não* tem variáveis locais que sombreiam as variáveis de instância da classe.

Dica de desempenho 8.1

O Java conserva armazenamento mantendo somente uma cópia de cada método por classe — esse método é invocado por cada objeto dessa classe. Cada objeto, por outro lado, tem sua própria cópia das variáveis de instância da classe. Cada método da classe utiliza implicitamente `this` para determinar o objeto específico da classe a manipular.

O método `main` (linhas 6 a 10) da classe `ThisTest` demonstra a classe `SimpleTime`. A linha 8 cria uma instância da classe `SimpleTime` e invoca seu construtor. A linha 9 invoca o método `buildString` do objeto e então exibe os resultados.

8.5 Estudo de caso da classe `Time`: construtores sobrecarregados

Como você sabe, é possível declarar seu próprio construtor a fim de especificar como objetos de uma classe devem ser inicializados. A seguir, demonstraremos uma classe com vários **construtores sobrecarregados** que permitem que objetos dessa classe sejam inicializados de diferentes maneiras. Para sobrecarregar construtores, simplesmente forneça múltiplas declarações de construtor com assinaturas diferentes.

Classe `Time2` com construtores sobrecarregados

O construtor padrão da classe `Time1` na Figura 8.1 inicializou `hour`, `minute` e `second` para seus valores `0` padrão (isto é, meia-noite na data/hora universal). O construtor padrão não permite que clientes da classe inicializem a data/hora com valores não zero específicos. A classe `Time2` (Figura 8.5) contém cinco construtores sobrecarregados que fornecem maneiras convenientes de inicializar objetos. Nesse programa, quatro dos construtores invocam um quinto, o qual, por sua vez, garante que o valor fornecido para `hour` está no intervalo de 0 a 23, e os valores para `minute` e `second` estão, cada um, no intervalo de 0 a 59. O compilador invoca o construtor apropriado correspondendo o número, tipos e a ordem dos tipos dos argumentos especificados na chamada do construtor com o número, tipos e a ordem dos tipos dos parâmetros especificados em cada declaração de construtor. A classe `Time2` também fornece os métodos *set* e *get* para cada variável de instância.

```
1  // Figura 8.5: Time2.java
2  // declaração da classe Time2 com construtores sobrecarregados.
3
4  public class Time2
5  {
6     private int hour; // 0 - 23
7     private int minute; // 0 - 59
8     private int second; // 0 - 59
9
10    // construtor sem argumento Time2:
11    // inicializa cada variável de instância para zero
12    public Time2()
13    {
14       this(0, 0, 0); // invoca o construtor com três argumentos
15    }
16
```

continua

continuação

```
17      // Construtor Time2: hora fornecida, minuto e segundo padronizados para 0
18      public Time2(int hour)
19      {
20         this(hour, 0, 0); // invoca o construtor com três argumentos
21      }
22
23      // Construtor Time2: hora e minuto fornecidos, segundo padronizado para 0
24      public Time2(int hour, int minute)
25      {
26         this(hour, minute, 0); // invoca o construtor com três argumentos
27      }
28
29      // Construtor Time2: hour, minute e second fornecidos
30      public Time2(int hour, int minute, int second)
31      {
32         if (hour < 0 || hour >= 24)
33            throw new IllegalArgumentException("hour must be 0-23");
34
35         if (minute < 0 || minute >= 60)
36            throw new IllegalArgumentException("minute must be 0-59");
37
38         if (second < 0 || second >= 60)
39            throw new IllegalArgumentException("second must be 0-59");
40
41         this.hour = hour;
42         this.minute = minute;
43         this.second = second;
44      }
45
46      // Construtor Time2: outro objeto Time2 fornecido
47      public Time2(Time2 time)
48      {
49         // invoca o construtor com três argumentos
50         this(time.getHour(), time.getMinute(), time.getSecond());
51      }
52
53      // Métodos set
54      // Configura um novo valor de tempo usando hora universal;
55      // valida os dados
56      public void setTime(int hour, int minute, int second)
57      {
58         if (hour < 0 || hour >= 24)
59            throw new IllegalArgumentException("hour must be 0-23");
60
61         if (minute < 0 || minute >= 60)
62            throw new IllegalArgumentException("minute must be 0-59");
63
64         if (second < 0 || second >= 60)
65            throw new IllegalArgumentException("second must be 0-59");
66
67         this.hour = hour;
68         this.minute = minute;
69         this.second = second;
70      }
71
72      // valida e configura a hora
73      public void setHour(int hour)
74      {
75         if (hour < 0 || hour >= 24)
76            throw new IllegalArgumentException("hour must be 0-23");
77
78         this.hour = hour;
79      }
80
81      // valida e configura os minutos
82      public void setMinute(int minute)
83      {
```

continua

continuação

```
84          if (minute < 0 || minute >= 60)
85             throw new IllegalArgumentException("minute must be 0-59");
86
87          this.minute = minute;
88       }
89
90       // valida e configura os segundos
91       public void setSecond(int second)
92       {
93          if (second < 0 || second >= 60)
94             throw new IllegalArgumentException("second must be 0-59");
95
96          this.second = second;
97       }
98
99       // Métodos get
100      // obtém valor da hora
101      public int getHour()
102      {
103         return hour;
104      }
105
106      // obtém valor dos minutos
107      public int getMinute()
108      {
109         return minute;
110      }
111
112      // obtém valor dos segundos
113      public int getSecond()
114      {
115         return second;
116      }
117
118      // converte em String no formato de data/hora universal (HH:MM:SS)
119      public String toUniversalString()
120      {
121         return String.format(
122            "%02d:%02d:%02d", getHour(), getMinute(), getSecond());
123      }
124
125      // converte em String no formato padrão de data/hora (H:MM:SS AM ou PM)
126      public String toString()
127      {
128         return String.format("%d:%02d:%02d %s",
129            ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
130            getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
131      }
132   } // fim da classe Time2
```

Figura 8.5 | Classe `Time2` com construtores sobrecarregados.

Construtores da classe `Time2` — chamando um construtor a partir de outro via `this`

As linhas 12 a 15 declaram um assim chamado **construtor sem argumento** que é invocado sem argumentos. Depois de você declarar quaisquer construtores em uma classe, o compilador *não* fornecerá um *construtor padrão.* Esse construtor sem argumento garante que os clientes da classe `Time2` podem criar objetos `Time2` com valores padrão. Esse construtor simplesmente inicializa o objeto como especificado no corpo do construtor. No corpo, introduzimos um uso da referência `this` que só é permitido como a *primeira* instrução no corpo de um construtor. A linha 14 usa `this` na sintaxe de chamada de método para invocar o construtor `Time2` que recebe três parâmetros (linhas 30 a 44) com valores de 0 para `hour`, `minute` e `second`. Utilizar a referência `this` como mostrado aqui é uma maneira popular de *reutilizar* código de inicialização fornecido por outro dos construtores da classe em vez de definir um código semelhante no corpo do construtor sem argumentos. Utilizamos essa sintaxe em quatro dos cinco construtores

Time2 para tornar a classe mais fácil de manter e modificar. Se for necessário alterar a maneira como objetos da classe Time2 são inicializados, somente o construtor que os outros construtores da classe chamam precisará ser modificado.

Erro comum de programação 8.2

É um erro de compilação se `this` *for utilizado no corpo de um construtor para chamar outros construtores da mesma classe se essa chamada não for a primeira instrução do construtor. Também é um erro de compilação se um método tentar invocar um construtor diretamente via* `this`.

As linhas 18 a 21 declaram um construtor Time2 com um parâmetro int único que representa a hour que é passada com 0 em minute e second para o construtor nas linhas 30 a 44. As linhas 24 a 27 declaram um construtor Time2 que recebe dois parâmetros int que representam a hour e minute, passados com 0 de second para o construtor nas linhas 30 a 44. Como ocorre com o construtor sem argumentos, cada um desses construtores invoca o construtor nas linhas 30 a 44 para minimizar a duplicação de código. As linhas 30 a 44 declaram o construtor Time2, que recebe três parâmetros int que representam hour, minute e second. Esse construtor valida e inicializa as variáveis de instância.

As linhas 47 a 51 declaram um construtor Time2 que recebe uma referência para outro objeto Time2. Os valores do argumento Time2 são passados para o construtor de três argumentos nas linhas 30 a 44 para inicializar hour, minute e second. A linha 50 poderia ter acessado diretamente os valores de hour, minute e second do argumento time com as expressões time.hour, time.minute e time.second — mesmo que hour, minute e second sejam declarados como variáveis private da classe Time2. Isso se deve a um relacionamento especial entre objetos da mesma classe. Veremos mais adiante por que é preferível usar os métodos *get*.

Observação de engenharia de software 8.5

Quando um objeto de uma classe contém uma referência a um outro objeto da mesma classe, o primeiro objeto pode acessar todos *os dados e métodos do segundo objeto (incluindo aqueles que são* `private`*).*

O método `setTime` *da classe* `Time2`

O método setTime (linhas 56 a 70) lança uma IllegalArgumentException (linhas 59, 62 e 65) se quaisquer argumentos do método estão fora do intervalo. Do contrário, ele define as variáveis de instância de Time2 como os valores de argumento (linhas 67 a 69).

Notas com relação aos construtores e métodos set e get da classe `Time2`

Os métodos *get* de Time2 são chamados ao longo de toda a classe. Em particular, os métodos toUniversalString e toString chamam os métodos getHour, getMinute e getSecond na linha 122 e nas linhas 129 e 130, respectivamente. Em cada caso, esses métodos poderiam ter acessado os dados privados da classe diretamente sem chamar os métodos *get*. Mas considere a possibilidade de alterar a representação da hora de três valores int (requerendo 12 bytes de memória) para um único valor int a fim de representar o número total de segundos que se passou desde a meia-noite (requerendo 4 bytes de memória). Se esse tipo de alteração fosse feito, apenas os corpos dos métodos que acessam os dados private diretamente precisariam ser alterados — especialmente, o construtor de três argumentos, o método setTime e os métodos *set* e *get* individuais para hour, minute e second. Não haveria necessidade de modificar o corpo dos métodos toUniversalString ou toString porque eles *não* acessam os dados diretamente. Projetar a classe dessa maneira reduz a probabilidade de erros de programação ao alterar a implementação da classe.

Da mesma forma, cada construtor Time2 pode incluir uma cópia das instruções adequadas a partir do construtor de três argumentos. Fazer isso pode ser um pouco mais eficiente, porque as chamadas extras do construtor são eliminadas. Mas *duplicar* instruções torna mais difícil alterar a representação interna dos dados da classe. Fazer com que os construtores Time2 chamem o construtor com três argumentos exige que quaisquer alterações na implementação do construtor de três argumentos sejam feitas apenas uma vez. Além disso, o compilador pode otimizar os programas removendo as chamadas para os métodos simples e substituindo-as pelo código expandido das suas declarações — uma técnica conhecida como **colocar o código em linha**, o que melhora o desempenho do programa.

Utilizando construtores sobrecarregados da classe `Time2`

A classe Time2Test (Figura 8.6) invoca os construtores Time2 sobrecarregados (linhas 8 a 12 e 24). A linha 8 invoca o construtor Time2 sem argumento. As linhas 9 a 12 demonstram como passar argumentos para os outros construtores Time2. A linha 9 invoca o construtor de argumento único que recebe um int nas linhas 18 a 21 da Figura 8.5. A linha 10 invoca o construtor de dois argumentos nas linhas 24 a 27 da Figura 8.5. A linha 11 invoca o construtor de três argumentos nas linhas 30 a 44 da Figura 8.5. A linha 12 invoca o construtor de argumento único que recebe um Time2 nas linhas 47 a 51 da Figura 8.5. Então, o aplicativo exibe as representações String de cada objeto Time2 para confirmar que ele foi inicializado adequadamente (linhas 15 a 19). A linha 24

tenta inicializar `t6` criando um novo objeto `Time2` e passando três valores *inválidos* para o construtor. Quando o construtor tenta usar o valor de hora inválido para inicializar `hour` do objeto, ocorre uma `IllegalArgumentException`. Capturamos essa exceção na linha 26 e exibimos a mensagem de erro, o que resulta na última linha da saída.

```
1  // Figura 8.6: Time2Test.java
2  // Construtores sobrecarregados utilizados para inicializar objetos Time2.
3
4  public class Time2Test
5  {
6     public static void main(String[] args)
7     {
8        Time2 t1 = new Time2(); // 00:00:00
9        Time2 t2 = new Time2(2); // 02:00:00
10       Time2 t3 = new Time2(21, 34); // 21:34:00
11       Time2 t4 = new Time2(12, 25, 42); // 12:25:42
12       Time2 t5 = new Time2(t4); // 12:25:42
13
14       System.out.println("Constructed with:");
15       displayTime("t1: all default arguments", t1);
16       displayTime("t2: hour specified; default minute and second", t2);
17       displayTime("t3: hour and minute specified; default second", t3);
18       displayTime("t4: hour, minute and second specified", t4);
19       displayTime("t5: Time2 object t4 specified", t5);
20
21       // tenta inicializar t6 com valores inválidos
22       try
23       {
24          Time2 t6 = new Time2(27, 74, 99); // valores inválidos
25       }
26       catch (IllegalArgumentException e)
27       {
28          System.out.printf("%nException while initializing t6: %s%n",
29             e.getMessage());
30       }
31    }
32
33    // exibe um objeto Time2 nos formatos de 24 horas e 12 horas
34    private static void displayTime(String header, Time2 t)
35    {
36       System.out.printf("%s%n   %s%n   %s%n",
37          header, t.toUniversalString(), t.toString());
38    }
39 } // fim da classe Time2Test
```

```
Constructed with:
t1: all default arguments
   00:00:00
   12:00:00 AM
t2: hour specified; default minute and second
   02:00:00
   2:00:00 AM
t3: hour and minute specified; default second
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: Time2 object t4 specified
   12:25:42
   12:25:42 PM
Exception while initializing t6: hour must be 0-23
```

Figura 8.6 | Construtores sobrecarregados utilizados para inicializar objetos `Time2`.

8.6 Construtores padrão e sem argumentos

Cada classe *deve* ter pelo menos *um* construtor. Se você não fornecer nenhuma declaração em uma classe, o compilador cria um *construtor padrão* que *não* recebe nenhum argumento quando é invocado. O construtor padrão inicializa as variáveis de instância com os valores iniciais especificados nas suas declarações ou com seus valores padrão (zero para tipos numéricos primitivos, `false` para valores `boolean` e `null` para referências). Na Seção 9.4.1, veremos que o construtor padrão também realiza outra tarefa.

Lembre-se de que, se a sua classe declarar construtores, o compilador *não* criará um construtor padrão. Nesse caso, você deve declarar um construtor sem argumento se uma inicialização padrão for necessária. Como ocorre com um construtor padrão, um construtor sem argumentos é invocado com parênteses vazios. O construtor `Time2` sem argumentos (linhas 12 a 15 da Figura 8.5) inicializa explicitamente um objeto `Time2` passando ao construtor de três argumentos 0 para cada parâmetro. Uma vez que 0 é o valor padrão para variáveis de instância `int`, nesse exemplo o construtor sem argumentos na verdade poderia ser declarado com um corpo vazio. Nesse caso, cada variável de instância receberia seu valor padrão quando o construtor sem argumentos fosse chamado. Se omitíssemos o construtor sem argumentos, clientes dessa classe não seriam capazes de criar um objeto `Time2` com a expressão `new Time2()`.

Dica de prevenção de erro 8.2

Certifique-se de que você não inclui um tipo de retorno na definição de um construtor. O Java permite que outros métodos da classe além dos seus construtores tenham o mesmo nome da classe e especifiquem tipos de retorno. Esses métodos não *são construtores e* não *serão chamados quando um objeto da classe for instanciado.*

Erro comum de programação 8.3

Um erro de compilação ocorre se um programa tenta inicializar um objeto de uma classe passando o número ou tipo errado de argumentos para o construtor da classe.

8.7 Notas sobre os métodos *Set* e *Get*

Como você sabe, os campos de uma classe `private` *só* podem ser manipulados pelos seus métodos. Uma manipulação típica talvez seja o ajuste do saldo bancário de um cliente (por exemplo, uma variável de instância `private` de uma classe `BankAccount`) por um método `computeInterest`. Métodos *set* também são comumente chamados **métodos modificadores**, porque eles geralmente *mudam* o estado de um objeto — isto é, *modificam* os valores das variáveis de instância. Os métodos *get* também são comumente chamados de **métodos de acesso** ou **métodos de consulta**.

Métodos set *e* get versus *dados* `public`

Parece que fornecer as capacidades de *set* e *get* é essencialmente o mesmo que tornar `public` as variáveis de instância da classe. Essa é uma das sutilezas que torna o Java tão desejável para a engenharia de software. Uma variável de instância `public` pode ser lida ou gravada por qualquer método que tem uma referência que contém a variável. Se uma variável de instância for declarada `private`, um método *get* `public` certamente permitirá que outros métodos a acessem, mas o método *get* pode *controlar* como o cliente pode acessá-la. Por exemplo, um método *get* poderia controlar o formato dos dados que ele retorna, e assim proteger o código do cliente na representação dos dados real. Um método *set* `public` pode, e deve, examinar cuidadosamente tentativas de modificar o valor da variável e lançar uma exceção, se necessário. Por exemplo, tentativas para definir o dia do mês como 37 ou peso de uma pessoa como um valor negativo devem ser rejeitadas. Portanto, embora os métodos *set* e *get* possam fornecer acesso a dados `private`, o acesso é restrito pela implementação dos métodos. Isso ajuda a promover uma boa engenharia de software.

Observação de engenharia de software 8.6

Classes nunca devem ter dados `public` *não constantes, mas declarar dados* `public static final` *permite disponibilizar as constantes para os clientes da sua classe. Por exemplo, a classe* `Math` *oferece as constantes* `final` `Math.E` *e* `Math.PI` `public static final`.

Dica de prevenção de erro 8.3

Não forneça constantes `public static final` *se é provável que os valores das constantes mudem nas versões futuras do seu software.*

Teste de validade em métodos set

Os benefícios da integridade de dados não são automaticamente simples porque as variáveis de instância são declaradas `private` — você deve fornecer a verificação de validade. Métodos *set* de uma classe poderiam determinar que foram feitas tentativas de atribuir dados inválidos a objetos da classe. Normalmente métodos *set* têm um tipo de retorno `void` e usam o tratamento de exceção para indicar tentativas de atribuir dados inválidos. Discutiremos o tratamento de exceção em detalhes no Capítulo 11.

Observação de engenharia de software 8.7

Se apropriado, forneça métodos `public` *para alterar e recuperar os valores de variáveis de instância* `private`*. Essa arquitetura ajuda a ocultar a implementação de uma classe dos seus clientes, o que aprimora a modificabilidade do programa.*

Dica de prevenção de erro 8.4

Usar métodos set *e* get *ajuda a criar classes que são mais fáceis de depurar e manter. Se apenas um método realizar uma tarefa particular, como configurar uma instância de variável em um objeto, é mais fácil depurar e manter a classe. Se a variável de instância não for configurada corretamente, o código que na verdade modifica a variável de instância estará localizado em um único método* set*. Seus esforços de depuração podem focalizar esse único método.*

Métodos predicados

Uma outra utilização comum para métodos de acesso é testar se uma condição é *verdadeira* ou *falsa* — esses métodos costumam ser chamados de **métodos predicados**. Um exemplo seria o método `isEmpty` da classe `ArrayList`, que retorna `true` se a `ArrayList` estiver vazia e `false` caso contrário. Um programa pode testar `isEmpty` antes de tentar ler outro item de uma `ArrayList`.

8.8 Composição

Uma classe pode ter referências a objetos de outras classes como membros. Isso é chamado **composição** e, às vezes, é referido como um **relacionamento *tem um***. Por exemplo, um objeto `AlarmClock` precisa saber a data/hora atual *e* a data/hora em que ele supostamente deve soar o alarme, por isso é razoável incluir *duas* referências a objetos `Time` em um objeto `AlarmClock`. Um carro *tem um* volante, um pedal de freio e um pedal de acelerador.

Classe `Date`

Esse exemplo de composição contém as classes `Date` (Figura 8.7), `Employee` (Figura 8.8) e `EmployeeTest` (Figura 8.9). A classe `Date` (Figura 8.7) declara as variáveis de instância `month`, `day` e `year` (linhas 6 a 8) para representar uma data. O construtor recebe três parâmetros `int`. As linhas 17 a 19 validam o `month` — se ele estiver fora do intervalo, as linhas 18 e 19 lançam uma exceção. As linhas 22 a 25 validam o `day`. Se o dia estiver incorreto com base no número de dias no `month` particular (exceto 29 de fevereiro, que exige testes especiais para anos bissextos), as linhas 24 e 25 lançam uma exceção. As linhas 28 a 31 realizam o teste de ano bissexto para fevereiro. Se o mês é fevereiro e o dia é 29 e o `year` não é um ano bissexto, as linhas 30 e 31 lançam uma exceção. Se nenhuma exceção for lançada, então as linhas 33 a 35 inicializam as variáveis de instância de Date e as linhas 37 e 38 geram a referência `this` como uma `String`. Como `this` é uma referência ao objeto `Date` atual, o método `toString` do objeto (linhas 42 a 45) é chamado *implicitamente* para obter a representação `String` do objeto. Nesse exemplo, vamos supor que o valor para `year` está correto — uma classe `Date` de força industrial também deve validar o ano.

```
1  // Figura 8.7: Date.java
2  // Declaração da classe Date.
3
4  public class Date
5  {
6     private int month; // 1-12
7     private int day; // 1-31 conforme o mês
8     private int year; // qualquer ano
9
10    private static final int[] daysPerMonth =
11       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
12
13    // construtor: confirma o valor adequado para o mês e dia dado o ano
14    public Date(int month, int day, int year)
15    {
```

continua

continuação

```
16          // verifica se mês está no intervalo
17          if (month <= 0 || month > 12)
18             throw new IllegalArgumentException(
19                "month (" + month + ") must be 1-12");
20
21          // verifica se day está no intervalo para month
22          if (day <= 0 ||
23             (day > daysPerMonth[month] && !(month == 2 && day == 29)))
24             throw new IllegalArgumentException("day (" + day +
25                ") out-of-range for the specified month and year");
26
27          // verifique no ano bissexto se o mês é 2 e o dia é 29
28          if (month == 2 && day == 29 && !(year % 400 == 0 ||
29                (year % 4 == 0 && year % 100 != 0)))
30             throw new IllegalArgumentException("day (" + day +
31                ") out-of-range for the specified month and year");
32
33          this.month = month;
34          this.day = day;
35          this.year = year;
36
37          System.out.printf(
38             "Date object constructor for date %s%n", this);
39       }
40
41       // retorna uma String no formato mês/dia/ano
42       public String toString()
43       {
44          return String.format("%d/%d/%d", month, day, year);
45       }
46    } // fim da classe Date
```

Figura 8.7 | Declaração de classe `Date`.

Classe `Employee`

A classe `Employee` (Figura 8.8) contém variáveis de instância `firstName`, `lastName`, `birthDate` e `hireDate`. Os membros `firstName` e `lastName` são referências a objetos `String`. Os membros `birthDate` e `hireDate` são referências a objetos `String`. Isso demonstra que uma classe pode conter como variáveis de instância referências a objetos de outras classes. O construtor `Employee` (linhas 12 a 19) recebe quatro parâmetros que representam o primeiro nome, sobrenome, data de nascimento e data de contratação. Os objetos referenciados pelos parâmetros são atribuídos às variáveis de instância do objeto `Employee`. Quando o método `toString` da classe `Employee` é chamado, ele retorna uma `String` contendo o nome do empregado e as representações de `String` dos dois objetos `Date`. Cada uma dessas `Strings` é obtida com uma chamada *implícita* ao método `toString` da classe `Date`.

```
1  // Figura 8.8: Employee.java
2  // Classe Employee com referência a outros objetos.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // construtor para inicializar nome, data de nascimento e data de contratação
12    public Employee(String firstName, String lastName, Date birthDate,
13       Date hireDate)
14    {
15       this.firstName = firstName;
16       this.lastName = lastName;
17       this.birthDate = birthDate;
18       this.hireDate = hireDate;
```

continua

continuação

```
19      }
20
21      // converte Employee em formato de String
22      public String toString()
23      {
24         return String.format("%s, %s  Hired: %s  Birthday: %s",
25            lastName, firstName, hireDate, birthDate);
26      }
27   } // fim da classe Employee
```

Figura 8.8 | A classe `Employee` com referência a outros objetos.

Classe `EmployeeTest`

A classe `EmployeeTest` (Figura 8.9) cria dois objetos `Date` para representar o aniversário e a data de contratação, respectivamente, de um `Employee`. A linha 10 cria um `Employee` e inicializa suas variáveis de instância passando para o construtor duas `Strings` (representando o primeiro e último nomes do `Employee`) e dois objetos `Date` (representando o aniversário e a data de contratação). A linha 12 invoca *implicitamente* o método `toString` de `Employee` para exibir os valores das suas variáveis de instância e demonstrar que o objeto foi inicializado adequadamente.

```
1   // Figura 8.9: EmployeeTest.java
2   // Demonstração de composição.
3
4   public class EmployeeTest
5   {
6      public static void main(String[] args)
7      {
8         Date birth = new Date(7, 24, 1949);
9         Date hire = new Date(3, 12, 1988);
10        Employee employee = new Employee("Bob", "Blue", birth, hire);
11
12        System.out.println(employee);
13     }
14  } // fim da classe EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Figura 8.9 | A demonstração de composição.

8.9 Tipos `enum`

Na Figura 6.8, apresentamos o tipo `enum` básico que define um conjunto de constantes representadas como identificadores únicos. Nesse programa, as constantes `enum` representaram o status do jogo. Nesta seção, discutimos o relacionamento entre tipos e classes `enum`. Como classes, todos os tipos `enum` são tipos *por referência*. Um tipo `enum` é declarado com uma **declaração `enum`**, uma lista separada por vírgulas de *constantes `enum`* — a declaração pode opcionalmente incluir outros componentes das classes tradicionais como construtores, campos e métodos (como você verá em breve). Cada declaração `enum` declara uma classe `enum` com as seguintes restrições:

1. constantes `enum` são *implicitamente* `final`.
2. constantes `enum` são *implicitamente* `static`.
3. Qualquer tentativa de criar um objeto de um tipo `enum` com um operador `new` resulta em um erro de compilação.

As constantes `enum` podem ser utilizadas em qualquer lugar em que constantes podem ser utilizadas, como nos rótulos `case` das instruções `switch` e para controlar instruções `for` aprimoradas.

Declarando variáveis de instância, um construtor e métodos em um tipo `enum`

A Figura 8.10 demonstra variáveis de instância, um construtor e métodos em um tipo `enum`. A declaração `enum` (linhas 5 a 37) contém duas partes — as constantes `enum` e os outros membros do tipo `enum`. A primeira parte (linhas 8 a 13) declara seis constantes.

Cada uma delas é opcionalmente seguida por argumentos que são passados para o **construtor enum** (linhas 20 a 24). Como os construtores que você viu nas classes, um construtor `enum` pode especificar qualquer número de parâmetros e pode ser sobrecarregado. Nesse exemplo, o construtor `enum` requer dois parâmetros `String`. Para inicializar adequadamente cada constante `enum`, ela é colocada entre parênteses contendo dois argumentos `String`. A segunda parte (linhas 16 a 36) declara os outros membros do tipo `enum` — duas variáveis de instância (linhas 16 e 17), um construtor (linhas 20 a 24) e dois métodos (linhas 27 a 30 e 33 a 36).

As linhas 16 e 17 declaram as variáveis de instância `title` e `copyrightYear`. Cada constante `enum` no tipo `Book` é na verdade um objeto do tipo `Book enum` que tem sua própria cópia das variáveis de instância `title` e `copyrightYear`. O construtor (linhas 20 a 24) recebe dois parâmetros `String`, um que especifica o título do livro e outro que especifica o ano dos direitos autorais. As linhas 22 e 23 atribuem esses parâmetros às variáveis de instância. As linhas 27 a 36 declaram dois métodos, que retornam o título de livro e o ano dos direitos autorais, respectivamente.

```
1  // Figura 8.10: Book.java
2  // Declarando um tipo enum com um construtor e campos de instância explícitos
3  // e métodos de acesso para esses campos
4
5  public enum Book
6  {
7     // declara constantes do tipo enum
8     JHTP("Java How to Program", "2015"),
9     CHTP("C How to Program", "2013"),
10    IW3HTP("Internet & World Wide Web How to Program", "2012"),
11    CPPHTP("C++ How to Program", "2014"),
12    VBHTP("Visual Basic How to Program", "2014"),
13    CSHARPHTP("Visual C# How to Program", "2014");
14
15    // campos de instância
16    private final String title; // título de livro
17    private final String copyrightYear; // ano dos direitos autorais
18
19    // construtor enum
20    Book(String title, String copyrightYear)
21    {
22       this.title = title;
23       this.copyrightYear = copyrightYear;
24    }
25
26    // acessor para título de campo
27    public String getTitle()
28    {
29       return title;
30    }
31
32    // acessor para o campo copyrightYear
33    public String getCopyrightYear()
34    {
35       return copyrightYear;
36    }
37 } // fim do enum Book
```

Figura 8.10 | Declarando um tipo `enum` com um construtor, campos de instância explícita e métodos acessores para esses campos.

Usando tipo enum Book

A Figura 8.11 testa o tipo `enum Book` e ilustra como iterar por um intervalo de constantes `enum`. Para cada `enum`, o compilador gera um método `static` chamado **values** (chamado na linha 12) que retorna um array das constantes do `enum` na ordem em que elas foram declaradas. As linhas 12 a 14 utilizam a instrução `for` aprimorada para exibir todas as constantes declaradas em `enum Book`. A linha 14 invoca os métodos `getTitle` e `getCopyrightYear` de `enum Book` para obter o título e o ano dos direitos autorais associado com a constante. Quando uma constante `enum` é convertida em uma `String` (por exemplo, `book` na linha 13), o identificador da constante é utilizado como a representação de `String` (por exemplo, `JHTP` para a primeira constante `enum`).

```
1  // Figura 8.11: EnumTest.java
2  // Testando o tipo enum Book.
3  import java.util.EnumSet;
4
5  public class EnumTest
6  {
7     public static void main(String[] args)
8     {
9        System.out.println("All books:");
10
11       // imprime todos os livros em enum Book
12       for (Book book : Book.values())
13          System.out.printf("%-10s%-45s%s%n", book,
14             book.getTitle(),book.getCopyrightYear());
15
16       System.out.printf("%nDisplay a range of enum constants:%n");
17
18       // imprime os primeiros quatro livros
19       for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTP))
20          System.out.printf("%-10s%-45s%s%n", book,
21             book.getTitle(),book.getCopyrightYear());
22    }
23 } // fim da classe EnumTest
```

```
All books:
JHTP      Java How to Program                          2015
CHTP      C How to Program                             2013
IW3HTP    Internet & World Wide Web How to Program     2012
CPPHTP    C++ How to Program                           2014
VBHTP     Visual Basic How to Program                  2014
CSHARPHTP Visual C# How to Program                     2014

Display a range of enum constants:
JHTP      Java How to Program                          2015
CHTP      C How to Program                             2013
IW3HTP    Internet & World Wide Web How to Program     2012
CPPHTP    C++ How to Program                           2014
```

Figura 8.11 | Testando o tipo `enum Book`.

As linhas 19 a 21 utilizam o método `static` **`range`** da classe **`EnumSet`** (declarado no pacote `java.util`) para exibir um intervalo das constantes do `enum Book`. O método `range` recebe dois parâmetros — as primeiras e as últimas constantes `enum` no intervalo — e retorna um `EnumSet` que contém todas as constantes entre essas duas constantes, inclusive. Por exemplo, a expressão `EnumSet.range(Book.JHTP, Book.CPPHTP)` retorna um `EnumSet` que contém `Book.JHTP`, `Book.CHTP`, `Book.IW3HTP` e `Book.CPPHTP`. A instrução `for` aprimorada pode ser utilizada com um `EnumSet`, assim como com um array, portanto as linhas 12 a 14 utilizam-na para exibir o título e o ano dos direitos autorais de cada livro na `EnumSet`. A classe `EnumSet` fornece vários outros métodos `static` para criar conjuntos de constantes `enum` do mesmo tipo `enum`.

Erro comum de programação 8.4

Em uma declaração `enum`, é um erro de sintaxe declarar constantes `enum` após construtores, campos e métodos do tipo `enum`.

8.10 Coleta de lixo

Cada objeto utiliza recursos do sistema, como memória. Precisamos de uma maneira disciplinada de retornar os recursos ao sistema quando eles não são mais necessários; caso contrário, podem ocorrer "vazamentos de recursos" que evitariam que eles fossem reutilizados pelo seu programa ou possivelmente por outros programas. A JVM executa **coleta de lixo** automática para recuperar a *memória* ocupada por objetos que não são mais usados. Quando *não* há *mais referências* a um objeto, o objeto é *marcado* para coleta de lixo. A coleta normalmente ocorre quando a JVM executa o **coletor de lixo**, o que pode não acontecer por um tempo, ou até mesmo absolutamente antes de um programa terminar. Assim, vazamentos de memória que são comuns em outras linguagens como C e C++ (porque a memória *não* é automaticamente reivindicada nessas linguagens) são *menos* prováveis em Java, mas alguns

ainda podem acontecer de maneiras sutis. Vazamentos de recursos além de vazamentos de memória também podem ocorrer. Por exemplo, um aplicativo pode abrir um arquivo no disco para modificar seu conteúdo — se o aplicativo não fechar o arquivo, ele deve terminar antes que qualquer outro aplicativo possa usar o arquivo.

Uma nota sobre o método `finalize` da classe `Object`

Toda classe no Java contém os métodos da classe `Object` (pacote `java.lang`), um dos quais é o método **`finalize`**. (Você aprenderá mais sobre a classe `Object` no Capítulo 9.) Você *nunca* deve usar o método `finalize`, porque ele pode causar muitos problemas e não há certeza se ele *alguma vez* será chamado antes de um programa terminar.

A intenção original de `finalize` era permitir que o coletor de lixo executasse a **faxina de término** em um objeto um pouco antes de reivindicar a memória do objeto. Agora, é considerada uma boa prática que qualquer classe que usa os recursos do sistema, como arquivos em disco, forneça um método que os programadores possam chamar para liberar os recursos quando eles não são mais necessários em um programa. Objetos `AutoClosable` reduzem a probabilidade de vazamentos de recursos ao usá-los com a instrução `try` com recursos. Como o próprio nome indica, um objeto `AutoClosable` é fechado automaticamente, depois que uma instrução `try` com recursos termina de usar o objeto. Discutiremos isso em mais detalhes na Seção 11.12.

Observação de engenharia de software 8.8

Muitas classes Java API (por exemplo, a classe `Scanner` e classes que leem arquivos ou gravam arquivos em disco) fornecem o método `close` ou `dispose`, que os programadores podem chamar para liberar os recursos quando eles não são mais necessários em um programa.

8.11 Membros da classe `static`

Cada objeto tem sua própria cópia de todas as variáveis de instância da classe. Em certos casos, apenas uma cópia de uma variável particular deve ser *compartilhada* por todos os objetos de uma classe. Um **campo `static`** — chamado **variável de classe** — é utilizado nesses casos. Uma variável `static` representa **informações de escopo de classe** — todos os objetos da classe compartilham os *mesmos* dados. A declaração de uma variável `static` inicia com a palavra-chave `static`.

Motivando `static`

Vamos motivar a necessidade de dados `static` dados com um exemplo. Suponha que tivéssemos um videogame com `Martians` e outras criaturas do espaço. Cada `Martian` tende a ser corajoso e disposto a atacar outras criaturas espaciais quando o `Martian` está ciente de que pelo menos cinco `Martians` estão presentes. Se menos de cinco `Martians` estiverem presentes, cada um deles torna-se covarde. Assim, cada `Martian` precisa conhecer o `martianCount`. Poderíamos dotar a classe `Martian` com `martianCount` como uma *variável de instância*. Se fizermos isso, então cada `Martian` terá *uma cópia separada* da variável de instância, e toda vez que criarmos um novo `Martian`, teremos de atualizar a variável de instância `martianCount` em cada objeto `Martian`. Isso desperdiça espaço com as cópias redundantes, desperdiça tempo com a atualização das cópias separadas e é propenso a erros. Em vez disso, declaramos `martianCount` como `static`, tornando `martianCount` dados de escopo de classe. Cada `Martian` pode ver o `martianCount` como se ele fosse uma variável de instância da classe `Martian`, mas somente *uma* cópia do `static` `martianCount` é mantida. Isso economiza espaço. Poupamos tempo fazendo com que o construtor `Martian` incremente o `static` `martianCount` — há somente uma cópia, assim não temos de incrementar cópias separadas de `martianCount` para cada objeto `Martian`.

Observação de engenharia de software 8.9

Utilize uma variável `static` quando todos os objetos de uma classe precisarem utilizar a mesma cópia da variável.

Escopo de classe

Variáveis estáticas têm *escopo de classe* — elas podem ser usadas em todos os métodos da classe. Podemos acessar membros `public static` de uma classe por meio de uma referência a qualquer objeto da classe ou qualificando o nome de membro com o nome de classe e um ponto (`.`), como em `Math.random()`. Membros da classe `private static` de uma classe podem ser acessados pelo código do cliente somente por métodos da classe. Realmente, *os membros da classe `static` existem mesmo quando não há nenhum objeto da classe* — eles estão disponíveis logo que a classe é carregada na memória em tempo de execução. Para acessar um membro `public static` quando não há nenhum objeto da classe (e mesmo se houver), prefixe o nome da classe e acrescente um ponto (`.`) ao membro `static`, como em `Math.PI`. Para acessar um membro `private static` quando não existem objetos da classe, forneça um método `public static` e chame-o qualificando seu nome com o nome da classe e um ponto.

Observação de engenharia de software 8.10

Variáveis e métodos de classe static existem e podem ser utilizados, mesmo se nenhum objeto dessa classe tiver sido instanciado.

Métodos `static` não podem acessar diretamente variáveis de instância e métodos de instância

Um método `static` *não pode* acessar as variáveis de instância e os métodos de instância de uma classe, porque um método `static` pode ser chamado mesmo quando nenhum objeto da classe foi instanciado. Pela mesma razão, a referência `this` *não pode* ser utilizada em um método `static`. A referência `this` deve se referir a um objeto específico da classe e, quando um método `static` é chamado, talvez não haja nenhum objeto da sua classe na memória.

Erro comum de programação 8.5

Um erro de compilação ocorre se um método `static` chamar um método de instância na mesma classe utilizando apenas o nome do método. De maneira semelhante, um erro de compilação ocorre se um método `static` tentar acessar uma variável de instância na mesma classe utilizando apenas o nome da variável.

Erro comum de programação 8.6

Referenciar `this` em um método `static` é um erro de compilação.

Monitorando o número de objetos Employee que foram criados

Nosso próximo programa declara duas classes — `Employee` (Figura 8.12) e `EmployeeTest` (Figura 8.13). A classe `Employee` declara uma variável `private static count` (Figura 8.12, linha 7) e o método `getCount public static` (linhas 36 a 39). A variável `count static` mantém uma contagem do número de objetos da classe `Employee` que foram criados até agora. A classe variável é inicializada como zero na linha 7. Se uma variável `static` *não* for inicializada, o compilador atribuirá um valor padrão — nesse caso `0`, o valor padrão para o tipo `int`.

```
1  // Figura 8.12: Employee.java
2  // Variável static utilizada para manter uma contagem do número de
3  // objetos Employee na memória.
4
5  public class Employee
6  {
7     private static int count = 0; // número de Employees criados
8     private String firstName;
9     private String lastName;
10
11    // inicializa Employee, adiciona 1 a static count e
12    // gera a saída de String indicando que o construtor foi chamado
13    public Employee(String firstName, String lastName)
14    {
15       this.firstName = firstName;
16       this.lastName = lastName;
17
18       ++count;  // incrementa contagem estática de empregados
19       System.out.printf("Employee constructor: %s %s; count = %d%n",
20          firstName, lastName, count);
21    }
22
23    // obtém o primeiro nome
24    public String getFirstName()
25    {
26       return firstName;
27    }
28
29    // obtém o último nome
```

continua

continuação

```
30       public String getLastName()
31       {
32          return lastName;
33       }
34
35       // método estático para obter valor de contagem de estática
36       public static int getCount()
37       {
38          return count;
39       }
40    } // fim da classe Employee
```

Figura 8.12 | Variável `static` utilizada para manter uma contagem do número de objetos `Employee` na memória.

Quando existem objetos `Employee`, a variável `count` pode ser usada em qualquer método de um objeto `Employee` — esse exemplo incrementa `count` no construtor (linha 18). O método `getCount` `public` `static` (linhas 36 a 39) retorna o número de objetos `Employee` que foram criados até agora. Quando não existem objetos da classe `Employee`, o código do cliente pode acessar a variável `count` chamando o método `getCount` pelo nome da classe, como em `Employee.getCount()`. Quando existem objetos, o método `getCount` também pode ser chamado por qualquer referência a um objeto `Employee`.

Boa prática de programação 8.1

Invoque cada método `static` utilizando o nome de classe e um ponto (.) para enfatizar que o método sendo chamado é um método `static`.

Classe `EmployeeTest`

O método `EmployeeTest` `main` (Figura 8.13) instancia dois objetos `Employee` (linhas 13 e 14). Quando cada construtor do objeto `Employee` é invocado, as linhas 15 e 16 da Figura 8.12 atribuem o primeiro nome e o sobrenome de `Employee` às variáveis de instância `firstName` e `lastName`. Essas duas instruções *não* criam cópias dos argumentos `String` originais. Na verdade, objetos `String` em Java são **imutáveis** — eles não podem ser modificados depois de criados. Portanto, é seguro ter *muitas* referências a um objeto `String`. Isso normalmente não é o caso para objetos da maioria das outras classes em Java. Se objetos `String` são imutáveis, você talvez se pergunte por que somos capazes de utilizar operadores + e += para concatenar objetos `String`. Na verdade, a concatenação de string resulta em um *novo* objeto `String` contendo os valores concatenados. Os objetos `String` originais *não* são modificados.

```
1  // Figura 8.13: EmployeeTest.java
2  // Demonstração do membro static.
3
4  public class EmployeeTest
5  {
6     public static void main(String[] args)
7     {
8        // mostra que a contagem é 0 antes de criar Employees
9        System.out.printf("Employees before instantiation: %d%n",
10          Employee.getCount());
11
12       // cria dois Employees; a contagem deve ser 2
13       Employee e1 = new Employee("Susan", "Baker");
14       Employee e2 = new Employee("Bob", "Blue");
15
16       // mostra que a contagem é 2 depois de criar dois Employees
17       System.out.printf("%nEmployees after instantiation:%n");
18       System.out.printf("via e1.getCount(): %d%n", e1.getCount());
19       System.out.printf("via e2.getCount(): %d%n", e2.getCount());
20       System.out.printf("via Employee.getCount(): %d%n",
21          Employee.getCount());
22
23       // obtém nomes de Employees
24       System.out.printf("%nEmployee 1: %s %s%nEmployee 2: %s %s%n",
```

continua

```
25            e1.getFirstName(), e1.getLastName(),
26            e2.getFirstName(), e2.getLastName());
27      }
28   } // fim da classe EmployeeTest
```

continuação

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Figura 8.13 | A demonstração do membro `static`.

Quando `main` termina, as variáveis `e1` e `e2` locais são descartadas — lembre-se de que uma variável local *só* existe até que o bloco em que ela é declarada conclui a execução. Como `e1` e `e2` eram as únicas referências aos objetos `Employee` criados nas linhas 13 e 14 (Figura 8.13), esses objetos são "marcados para a coleta de lixo" quando `main` termina.

Em um aplicativo típico, o coletor de lixo *pode* eventualmente reivindicar a memória para todos os objetos que são marcados para a coleta de lixo. Se quaisquer objetos não forem reivindicados antes de o programa terminar, o sistema operacional irá reivindicar a memória usada pelo programa. A JVM *não* garante quando, ou mesmo se, o coletor de lixo será executado. Quando ela garante, é possível que nenhum objeto ou apenas um subconjunto dos objetos marcados serão coletados.

8.12 Importação `static`

Na Seção 6.3, vimos os campos e métodos `static` da classe `Math`. Acessamos campos e *métodos* `static` da classe `Math` e precedendo cada um com o nome da classe `Math` e um ponto (`.`). A declaração de **importação `static`** permite importar os membros `static` de uma interface ou classe para que você possa acessá-los por meio dos *nomes não qualificados* na sua classe — isto é, um ponto (`.`) e o nome da classe *não* são necessários ao usar um membro `static` importado.

Importando formulários `static`

Uma declaração de importação `static` tem duas formas — uma que importa um membro `static` particular (conhecido como **importação `static` simples**) e outra que importa *todos* os membros `static` de uma classe (conhecido como **importação `static` por demanda**). A sintaxe a seguir importa um membro `static` particular:

```
import static nomeDoPacote.NomeDaClasse.nomeDoMembroStatic;
```

onde *nomeDoPacote* é o pacote da classe (por exemplo, `java.lang`), *NomeDaClasse* é o nome da classe (por exemplo, `Math`) e *nomeDoMembroStatic* é o nome do campo ou método `static` (por exemplo, `PI` ou `abs`). A sintaxe a seguir importa *todos* os membros `static` de uma classe:

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

O asterisco (`*`) indica que *todos* os membros `static` da classe especificada devem estar disponíveis para uso no arquivo. Declarações de importação `static` importam *somente* os membros da classe `static`. Instruções `import` regulares devem ser utilizadas para especificar as classes utilizadas em um programa.

Demonstrando importação `static`

A Figura 8.14 demonstra uma importação `static`. A linha 3 é uma declaração de importação `static` que importa *todos* os campos e métodos `static` da classe `Math` no pacote `java.lang`. As linhas 9 a 12 acessam os campos `E` (linha 11) e `PI` (linha 12) `static` da classe `Math` e os métodos `sqrt` (linha 9) e `ceil` (linha 10) `static` *sem* preceder os nomes de campo ou nomes de método com um ponto e o nome da classe `Math`.

Erro comum de programação 8.7

Um erro de compilação ocorre se um programa tentar importar métodos `static` que têm a mesma assinatura ou campos `static` que têm o mesmo nome proveniente de duas ou mais classes.

```
1  // Figura 8.14: StaticImportTest.java
2  // Importação static dos métodos da classe Math.
3  import static java.lang.Math.*;
4
5  public class StaticImportTest
6  {
7     public static void main(String[] args)
8     {
9        System.out.printf("sqrt(900.0) = %.1f%n", sqrt(900.0));
10       System.out.printf("ceil(-9.8) = %.1f%n", ceil(-9.8));
11       System.out.printf("E = %f%n", E);
12       System.out.printf("PI = %f%n", PI);
13    }
14 } // fim da classe StaticImportTest
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Figura 8.14 | Importação `static` dos métodos da classe `Math`.

8.13 Variáveis de instância `final`

O **princípio do menor privilégio** é fundamental para uma boa engenharia de software. No contexto de um aplicativo, ele declara que deve ser concedido ao código somente a quantidade de privilégio e acesso que ele precisa para realizar sua tarefa designada, mas não mais que isso. Isso torna seus programas mais robustos evitando que o código modifique acidentalmente (ou maliciosamente) os valores das variáveis e chame métodos que *não* deveriam estar acessíveis.

Veremos como esse princípio se aplica a variáveis de instância. Algumas delas precisam ser *modificáveis* e algumas não. Você pode utilizar a palavra-chave `final` para especificar o fato de que uma variável *não* é modificável (isto é, é uma *constante*) e que qualquer tentativa de modificá-la é um erro. Por exemplo,

```
private final int INCREMENT;
```

declara uma variável de instância `final` `INCREMENT` (constante) do tipo `int`. Essas variáveis podem ser inicializadas quando elas são declaradas. Se não forem, elas *devem* ser inicializadas em cada construtor da classe. Inicializar constantes em construtores permite que cada objeto da classe tenha um valor diferente para a constante. Se uma variável `final` *não* é inicializada na sua declaração ou em cada construtor, ocorre um erro de compilação.

Observação de engenharia de software 8.11

Declarar uma variável de instância como `final` ajuda a impor o princípio do menor privilégio. Se uma variável de instância não deve ser modificada, declare-a como `final` para evitar modificação. Por exemplo, na Figura 8.8, as variáveis de instância `firstName`, `lastName`, `birthDate` e `hireDate` nunca são modificadas depois que elas são inicializadas, então elas devem ser declaradas `final`. Vamos aplicar essa prática em todos os programas daqui para a frente. Veremos os benefícios adicionais de `final` no Capítulo 23, "Concorrência".

Erro comum de programação 8.8

Tentar modificar uma variável de instância `final` depois que é ela inicializada é um erro de compilação.

Dica de prevenção de erro 8.5

Tentativas de modificar uma variável de instância `final` são capturadas em tempo de compilação em vez de causarem erros em tempo de execução. Sempre é preferível retirar bugs em tempo de compilação, se possível, em vez de permitir que passem para o tempo de execução (onde experiências descobriram que o reparo é frequentemente muito mais caro).

Observação de engenharia de software 8.12

Um campo `final` também deve ser declarado `static` se ele for inicializado na sua declaração para um valor que é o mesmo para todos os objetos da classe. Após essa inicialização, seu valor nunca pode mudar. Portanto, não precisamos de uma cópia separada do campo para cada objeto da classe. Criar o campo `static` permite que todos os objetos da classe compartilhem o campo `final`.

8.14 Acesso de pacote

Se nenhum modificador de acesso (`public`, `protected` ou `private` — `protected` será discutido no Capítulo 9) for especificado para um método ou variável quando esse método ou variável é declarado em uma classe, o método ou variável será considerado como tendo **acesso de pacote**. Em um programa que consiste em uma declaração de classe, isso não tem nenhum efeito específico. Entretanto, se um programa utilizar *múltiplas* classes no *mesmo* pacote (isto é, um grupo de classes relacionadas), essas classes poderão acessar diretamente os membros de acesso de pacote de outras classes por meio de referências a objetos das classes apropriadas, ou no caso de membros `static`, por meio do nome de classe. O acesso de pacote é raramente usado.

A Figura 8.15 demonstra o acesso de pacote. O aplicativo contém duas classes em um arquivo de código-fonte — a classe `PackageDataTest`, que contém `main` (linhas 5 a 21), e a classe `PackageData` (linhas 24 a 41). As classes no mesmo arquivo fonte são parte do mesmo pacote. Consequentemente, a classe `PackageDataTest` pode modificar os dados de acesso de pacote dos objetos `PackageData`. Ao compilar esse programa, o compilador produz dois arquivos `.class` separados — `PackageDataTest.class` e `PackageData.class`. O compilador coloca os dois arquivos `.class` no mesmo diretório. Você também pode colocar a classe `PackageData` (linhas 24 a 41) em um arquivo de código-fonte separado.

Na declaração da classe `PackageData`, as linhas 26 e 27 declaram as variáveis de instância `number` e `string` sem modificadores de acesso — portanto, elas são variáveis de instância de acesso de pacote. O método `main` da classe `PackageDataTest` cria uma instância da classe `PackageData` (linha 9) para demonstrar a capacidade de modificar as variáveis de instância `PackageData` diretamente (como mostrado nas linhas 15 e 16). Os resultados da modificação podem ser vistos na janela de saída.

```
1  // Figura 8.15: PackageDataTest.java
2  // Membros de acesso de pacote de uma classe permanecem acessíveis a outras classes
3  // no mesmo pacote.
4
5  public class PackageDataTest
6  {
7     public static void main(String[] args)
8     {
9        PackageData packageData = new PackageData();
10
11       // gera saída da representação String de packageData
12       System.out.printf("After instantiation:%n%s%n", packageData);
13
14       // muda os dados de acesso de pacote no objeto packageData
15       packageData.number = 77;
16       packageData.string = "Goodbye";
17
18       // gera saída da representação String de packageData
19       System.out.printf("%nAfter changing values:%n%s%n", packageData);
20    }
21 } // fim da classe PackageDataTest
22
23 // classe com variáveis de instância de acesso de pacote
24 class PackageData
25 {
26    int number; // variável de instância de acesso de pacote
27    String string; // variável de instância de acesso de pacote
28
29    // construtor
30    public PackageData()
31    {
32       number = 0;
33       string = "Hello";
34    }
35
36    // retorna a representação String do objeto PackageData
37    public String toString()
38    {
39       return String.format("number: %d; string: %s", number, string);
40    }
41 } // fim da classe PackageData
```

continua

continuação

```
After instantiation:
number: 0; string: Hello

After changing values:
number: 77; string: Goodbye
```

Figura 8.15 | Os membros de acesso de pacote de uma classe permanecem acessíveis a outras classes no mesmo pacote.

8.15 Usando `BigDecimal` para cálculos monetários precisos

Nos capítulos anteriores, demonstramos cálculos monetários utilizando valores do tipo `double`. No Capítulo 5, discutimos o fato de que alguns valores `double` são representados *aproximadamente*. Qualquer aplicação que requer cálculos precisos de ponto flutuante — como aplicações financeiras — deve usar a classe **`BigDecimal`** (do pacote **`java.math`**).

Cálculos de juros usando `BigDecimal`

A Figura 8.16 reimplementa o exemplo de cálculo de juros da Figura 5.6 usando objetos da classe `BigDecimal` para realizar os cálculos. Também introduzimos a classe **`NumberFormat`** (pacote **`java.text`**) para formatar valores numéricos como `Strings` *específicas de localidade*, por exemplo, na localidade dos EUA, o valor 1.234,56, seria formatado como `"1,234.56"`, enquanto em muitas localidades europeias (ou brasileiras) ele seria formatado como "`1.234,56`".

```
1  // Interest.java
2  // Cálculos de juros compostos com BigDecimal.
3  import java.math.BigDecimal;
4  import java.text.NumberFormat;
5
6  public class Interest
7  {
8     public static void main(String args[])
9     {
10       // quantidade principal inicial antes dos juros
11       BigDecimal principal = BigDecimal.valueOf(1000.0);
12       BigDecimal rate = BigDecimal.valueOf(0.05); // taxa de juros
13
14       // exibe cabeçalhos
15       System.out.printf("%s%20s%n", "Year", "Amount on deposit");
16
17       // calcula quantidade de depósito para cada um dos dez anos
18       for (int year = 1; year <= 10; year++)
19       {
20          // calcula nova quantidade durante ano especificado
21          BigDecimal amount =
22             principal.multiply(rate.add(BigDecimal.ONE).pow(year));
23
24          // exibe o ano e a quantidade
25          System.out.printf("%4d%20s%n", year,
26             NumberFormat.getCurrencyInstance().format(amount));
27       }
28    }
29 } // fim da classe Interest
```

```
Year   Amount on deposit
   1         $1,050.00
   2         $1,102.50
   3         $1,157.62
   4         $1,215.51
   5         $1,276.28
   6         $1,340.10
   7         $1,407.10
   8         $1,477.46
   9         $1,551.33
  10         $1,628.89
```

Figura 8.16 | Cálculos de juros compostos com `BigDecimal`.

Criando objetos `BigDecimal`

As linhas 11 e 12 declaram e inicializam variáveis `BigDecimal` `rate` e `principal` e usam o método `BigDecimal` `static` **`valueOf`**, que recebe um argumento `double` e retorna um objeto `BigDecimal` que representa o valor *exato* especificado.

Realizando os cálculos de juros com `BigDecimal`

As linhas 21 e 22 realizam o cálculo de juros utilizando métodos `BigDecimal` **`multiply`** , **`add`** e **`pow`**. A expressão na linha 22 é avaliada desta maneira:

1. Primeiro, a expressão `rate.add(BigDecimal.ONE)` adiciona `1` a `rate` para produzir um `BigDecimal` contendo `1.05` — isso é equivalente a `1.0 + rate` na linha 19 da Figura 5.6. A constante **`ONE`** `BigDecimal` representa o valor `1`. A classe `BigDecimal` também fornece as constantes **`ZERO`** (`0`) e **`TEN`** (`10`) comumente utilizadas.
2. Então, o método `BigDecimal` `pow` é chamado no resultado anterior para elevar `1.05` à potência `year` — isso é equivalente a passar `1.0 + rate` e `year` para o método `Math.pow` na linha 19 da Figura 5.6.
3. Por fim, chamamos o método `BigDecimal` `multiply` no objeto `principal` passando o resultado anterior como o argumento. Isso retorna um `BigDecimal` que representa o valor no depósito no final do `year` especificado.

Como a expressão `rate.add(BigDecimal.ONE)` produz o mesmo valor em cada iteração do loop, poderíamos simplesmente inicializar a taxa para `1.05` na linha 12; mas optamos por simular os cálculos precisos que utilizamos na linha 19 da Figura 5.6.

Formatando valores de moeda com `NumberFormat`

Durante cada iteração do loop, a linha 26

```
NumberFormat.getCurrencyInstance().format(amount)
```

é avaliada como a seguir:

1. Primeiro, a expressão usa o método **`getCurrencyInstance`** `static` de `NumberFormat` para obter um `NumberFormat` que é pré-configurado para formatar valores numéricos como `String`s de moedas específicas da localidade, por exemplo, na localidade nos EUA, o valor numérico 1.628,89 é formatado como $ 1,628.89. Formatação específica da localidade é uma parte importante da **internacionalização** — o processo de personalização dos seus aplicativos para várias localidades e idiomas falados dos usuários.
2. Então, a expressão invoca o método `NumberFormat` **`format`** (no objeto retornado por `getCurrencyInstance`) para realizar a formatação do valor `amount`. O método `format` então retorna a representação `String` específica da localidade, arredondada para dois dígitos à direita do ponto decimal.

Arredondando valores `BigDecimal`

Além de cálculos precisos, `BigDecimal` também lhe dá controle sobre como os valores são arredondados — por padrão, todos os cálculos são exatos e *nenhum* arredondamento ocorre. Se você não especificar como arredondar valores `BigDecimal` e um determinado valor não pode ser representado exatamente — como o resultado de 1 dividido por 3, que é 0,3333333... — ocorre uma `ArithmeticException`.

Embora não façamos isso nesse exemplo, você pode especificar o *modo de arredondamento* para `BigDecimal` fornecendo um objeto `MathContext` (pacote `java.math`) para o construtor da classe `BigDecimal` ao criar um `BigDecimal`. Você também pode fornecer um `MathContext` para vários métodos `BigDecimal` que realizam os cálculos. A classe `MathContext` contém vários objetos `MathContext` pré-configurados, os quais podem ser vistos em

```
http://docs.oracle.com/javase/7/docs/api/java/math/MathContext.html
```

Por padrão, cada `MathContext` pré-configurado usa o chamado "arredondamento contábil" como explicado para a constante `HALF_EVEN` `RoundingMode` em:

```
http://docs.oracle.com/javase/7/docs/api/java/math/RoundingMode.html#HALF_EVEN
```

Escalonando valores `BigDecimal`

O escalonamento de um `BigDecimal` é o número de dígitos à direita do ponto decimal. Se você precisa de um `BigDecimal` arredondado para um dígito específico, chame o método `BigDecimal` `setScale`. Por exemplo, a seguinte expressão retorna um `BigDecimal` com dois dígitos à direita do ponto decimal e usa arredondamento contábil:

```
amount.setScale(2, RoundingMode.HALF_EVEN)
```

8.16 (Opcional) Estudo de caso de GUIs e imagens gráficas: utilizando objetos com imagens gráficas

A maioria dos elementos gráficos que você viu até agora não varia com cada execução do programa. O Exercício 6.2 da Seção 6.13 solicitou que você criasse um programa que gerasse formas e cores aleatoriamente. Naquele exercício, o desenho foi alterado sempre que o sistema chamou `paintComponent`. Para criar um desenho mais consistente que permaneça idêntico todas as vezes que é desenhado, devemos armazenar informações sobre as formas exibidas, de modo que possamos reproduzi-las toda vez que o sistema chamar `paintComponent`. Para fazer isso, criaremos um conjunto de classes de forma que armazene informações sobre cada forma. Tornaremos essas classes "inteligentes", permitindo que os objetos dessas classes desenhem eles mesmos usando um objeto `Graphics`.

Classe `MyLine`

A Figura 8.17 declara a classe `MyLine`, que tem todas essas capacidades. A classe `MyLine` importa as classes `Color` e `Graphics` (linhas 3 a 4). As linhas 8 a 11 declaram as variáveis de instância para as coordenadas das extremidades necessárias para desenhar uma linha, e a linha 12 declara a variável de instância que armazena a cor da linha. O construtor nas linhas 15 a 22 recebe cinco parâmetros, um para cada variável de instância que ele inicializa. O método `draw` nas linhas 25 a 29 requer um objeto `Graphics` e o utiliza para desenhar a linha na cor apropriada e nos pontos finais.

```
1  // Figura 8.17: MyLine.java
2  // A classe MyLine representa uma linha.
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class MyLine
7  {
8     private int x1; // coordenada x da primeira extremidade final
9     private int y1; // coordenada y da primeira extremidade final
10    private int x2; // coordenada x da segunda extremidade final
11    private int y2; // coordenada y da segunda extremidade final
12    private Color color; // atribui uma cor a essa linha
13
14    // construtor com valores de entrada
15    public MyLine(int x1, int y1, int x2, int y2, Color color)
16    {
17       this.x1 = x1;
18       this.y1 = y1;
19       this.x2 = x2;
20       this.y2 = y2;
21       this.color = color;
22    }
23
24    // Desenha a linha na cor especificada
25    public void draw(Graphics g)
26    {
27       g.setColor(color);
28       g.drawLine(x1, y1, x2, y2);
29    }
30 } // fim da classe MyLine
```

Figura 8.17 | Classe `MyLine` representa uma linha.

Classe `DrawPanel`

Na Figura 8.18, declaramos a classe `DrawPanel`, que irá gerar objetos aleatórios da classe `MyLine`. A linha 12 declara o array `lines` de `MyLine` para armazenar as linhas a desenhar. Dentro do construtor (linhas 15 a 37), a linha 17 configura a cor de segundo plano como `Color.WHITE`. A linha 19 cria o array com um comprimento aleatório entre 5 e 9. O loop nas linhas 22 a 36 cria um novo `MyLine` para cada elemento no array. As linhas 25 a 28 geram coordenadas aleatórias para as extremidades finais da linha, e as linhas 31 e 32 geram uma cor aleatória para a linha. A linha 35 cria um novo objeto `MyLine` com os valores aleatoriamente gerados e o armazena no array. O método `paintComponent` itera pelos objetos `MyLine` no array `lines` utilizando uma instrução `for` aprimorada (linhas 45 e 46). Cada iteração chama o método `draw` do objeto `MyLine` atual e passa para ele o objeto `Graphics` para desenhar no painel.

```
1  // Figura 8.18: DrawPanel.java
2  // Programa que utiliza a classe MyLine
3  // para desenhar linhas aleatórias.
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import java.security.SecureRandom;
7  import javax.swing.JPanel;
8
9  public class DrawPanel extends JPanel
10 {
11    private SecureRandom randomNumbers = new SecureRandom();
12    private MyLine[] lines; // array de linhas
13
14    // construtor, cria um painel com formas aleatórias
15    public DrawPanel()
16    {
17       setBackground(Color.WHITE);
18
19       lines = new MyLine[5 + randomNumbers.nextInt(5)];
20
21       // cria linhas
22       for (int count = 0; count < lines.length; count++)
23       {
24          // gera coordenadas aleatórias
25          int x1 = randomNumbers.nextInt(300);
26          int y1 = randomNumbers.nextInt(300);
27          int x2 = randomNumbers.nextInt(300);
28          int y2 = randomNumbers.nextInt(300);
29
30          // gera uma cor aleatória
31          Color color = new Color(randomNumbers.nextInt(256),
32             randomNumbers.nextInt(256), randomNumbers.nextInt(256));
33
34          // adiciona a linha à lista de linhas a ser exibida
35          lines[count] = new MyLine(x1, y1, x2, y2, color);
36       }
37    }
38
39    // para cada array de forma, desenha as formas individuais
40    public void paintComponent(Graphics g)
41    {
42       super.paintComponent(g);
43
44       // desenha as linhas
45       for (MyLine line : lines)
46          line.draw(g);
47    }
48 } // fim da classe DrawPanel
```

Figura 8.18 | O programa que usa a classe `MyLine` para desenhar linhas aleatórias.

Classe `TestDraw`

A classe `TestDraw` na Figura 8.19 configura uma nova janela para exibir nosso desenho. Como estamos configurando as coordenadas para as linhas somente uma vez no construtor, o desenho não muda se `paintComponent` for chamado para atualizar o desenho na tela.

```
1  // Figura 8.19: TestDraw.java
2  // Criando um JFrame para exibir um DrawPanel.
3  import javax.swing.JFrame;
4
5  public class TestDraw
6  {
```

continua

continuação

```
7       public static void main(String[] args)
8       {
9          DrawPanel panel = new DrawPanel();
10         JFrame app = new JFrame();
11
12         app.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13         app.add(panel);
14         app.setSize(300, 300);
15         app.setVisible(true);
16      }
17   } // fim da classe TestDraw
```

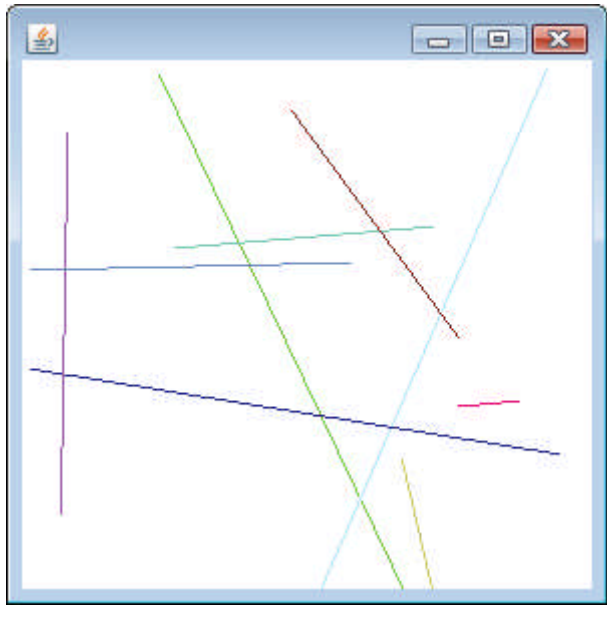


Figura 8.19 | Criando um `JFrame` para exibir um `DrawPanel`.

Exercício do estudo de caso GUI e imagens gráficas

8.1 Estenda o programa das figuras 8.17 a 8.19 para desenhar aleatoriamente retângulos e ovais. Crie as classes `MyRectangle` e `MyOval`. Essas duas classes devem incluir as coordenadas *x1*, *y1*, *x2*, *y2*, uma cor e um flag `boolean` para determinar se a forma é preenchida. Declare um construtor em cada classe com argumentos para inicializar todas as variáveis de instância. Para ajudar a desenhar retângulos e ovais, cada classe deve fornecer os métodos `getUpperLeftX`, `getUpperLeftY`, `getWidth` e `getHeight`, que calculam a coordenada *x* superior esquerda, a coordenada *y* superior esquerda e a largura e altura, respectivamente. A coordenada *x* superior esquerda é a menor dos dois valores da coordenada *x*, a coordenada *y* superior esquerda é a menor dos valores dois da coordenada *y*, a largura é o valor absoluto da diferença entre os dois valores da coordenada *x* e a altura é o valor absoluto da diferença entre os dois valores das coordenadas *y*.

A classe `DrawPanel`, que estende `JPanel` e trata a criação das formas, deve declarar três arrays, um para cada tipo de forma. O comprimento de cada array deve ser um número aleatório entre 1 e 5. O construtor da classe `DrawPanel` preencherá cada um dos arrays com formas de posição aleatória, tamanho, cor e preenchimento.

Além disso, modifique todas as três classes de forma a incluir o seguinte:

a) Um construtor sem argumentos que configura as coordenadas da forma como 0, a cor da forma como `Color.BLACK` e a propriedade preenchida como `false` (`MyRectangle` e `MyOval` somente).

b) Métodos *set* para as variáveis de instância em cada classe. Os métodos que configuram um valor de coordenada devem verificar se o argumento é maior ou igual a zero antes de configurar a coordenada — se não for, devem configurar a coordenada como zero. O construtor deve chamar os métodos *set* em vez de inicializar as variáveis locais diretamente.

c) Os métodos *get* para as variáveis de instância em cada classe. O método `draw` deve referenciar as coordenadas pelos métodos *get* em vez de acessá-los diretamente.

8.17 Conclusão

Neste capítulo, apresentamos conceitos adicionais sobre classes. O estudo de caso da classe `Time` mostrou uma declaração de classe completa que consiste em dados `private`, construtores `public` sobrecarregados para flexibilidade da inicialização, métodos *set* e *get* para manipular os dados da classe e métodos que retornaram representações de `String` de um objeto `Time` em duas formas

diferentes. Você também aprendeu que cada classe pode declarar um método `toString` que retorna uma representação `String` de um objeto da classe e que o método `toString` pode ser chamado implicitamente sempre que um objeto de uma classe aparece no código onde se espera uma `String`. Mostramos como usar `throw` para lançar uma exceção a fim de indicar que um problema ocorreu.

Você aprendeu que a referência `this` é usada implicitamente nos métodos de instância de uma classe para acessar as variáveis de instância e outros métodos de instância da classe. Você também viu utilizações explícitas da referência `this` para acessar os membros da classe (incluindo campos sombreados) e como utilizar a palavra-chave `this` em um construtor para chamar um outro construtor da classe.

Discutimos as diferenças entre construtores padrão fornecidos pelo compilador e construtores sem argumentos fornecidos pelo programador. Você aprendeu que uma classe pode ter referências a objetos de outras classes como membros — um conceito conhecido como composição. Você aprendeu mais sobre tipos `enum` e como eles podem ser utilizados para criar um conjunto de constantes para uso em um programa. Discutimos a capacidade da coleta de lixo do Java e como ela reivindica (inesperadamente) a memória de objetos que não são mais utilizados. O capítulo explicou a motivação da utilização de campos `static` em uma classe e demonstrou como declarar e utilizar campos e métodos `static` nas suas próprias classes. Você também aprendeu a declarar e inicializar variáveis `final`.

Você aprendeu que campos declarados sem um modificador de acesso têm acesso de pacote por padrão. Você viu o relacionamento entre classes no mesmo pacote, que permite a cada classe em um pacote acessar os membros de acesso de pacote de outras classes no pacote. Por fim, demonstramos como usar a classe `BigDecimal` para realizar cálculos monetários precisos.

No próximo capítulo, você aprenderá um aspecto importante da programação orientada a objetos em Java — a herança. Veremos que todas as classes em Java são relacionadas por herança, direta ou indiretamente à classe chamada `Object`. Você também entenderá como os relacionamentos entre classes permitem construir aplicativos mais poderosos.

Resumo

Seção 8.2 Estudo de caso da classe `Time`

- Os métodos `public` de uma classe também são conhecidos como os serviços `public` ou interface `public` da classe. Eles apresentam aos clientes da classe uma visualização dos serviços fornecidos.
- Membros `private` de uma classe não são acessíveis aos clientes.
- O método `static format` da classe `String` é semelhante ao método `System.out.printf`, exceto que `format` retorna uma `String` formatada em vez de exibi-la em uma janela de comando.
- Todos os objetos em Java têm um método `toString` que retorna uma representação `String` do objeto. O método `toString` é chamado implicitamente quando um objeto aparece no código onde uma `String` é necessária.

Seção 8.3 Controlando o acesso a membros

- Os modificadores de acesso `public` e `private` controlam o acesso às variáveis e métodos de uma classe.
- O principal propósito dos métodos `public` é apresentar para os clientes da classe uma visualização dos serviços fornecidos. Os clientes não precisam se preocupar com a forma como a classe realiza suas tarefas.
- Variáveis `private` e métodos `private` de uma classe (isto é, os detalhes de implementação) não são acessíveis aos clientes.

Seção 8.4 Referenciando membros do objeto atual com a referência `this`

- Um método de instância de um objeto utiliza implicitamente a palavra-chave `this` para referenciar variáveis de instância e outros métodos do objeto. A palavra-chave `this` também pode ser utilizada explicitamente.
- O compilador produz um arquivo separado com a extensão `.class` para cada classe compilada.
- Se uma variável local tiver o mesmo nome que o campo de uma classe, a variável local sombreia o campo. Você pode utilizar a referência `this` para referenciar o campo sombreado explicitamente.

Seção 8.5 Estudo de caso da classe `Time`: construtores sobrecarregados

- Construtores sobrecarregados permitem que objetos de uma classe sejam inicializados de diferentes maneiras. O compilador diferencia os construtores sobrecarregados por suas assinaturas.
- Para chamar um construtor de uma classe a partir de outro da mesma classe, use a palavra-chave `this` seguida por parênteses contendo os argumentos do construtor. Se utilizado, essa chamada de construtor deve aparecer como a primeira instrução no corpo do construtor.

Seção 8.6 Construtores padrão e sem argumentos

- Se nenhum construtor for fornecido em uma classe, o compilador cria um construtor padrão.
- Se uma classe declarar construtores, o compilador não criará um construtor padrão. Nesse caso, você deve declarar um construtor sem argumento se a inicialização padrão for necessária.

Seção 8.7 Notas sobre os métodos Set e Get

- Os métodos *set* são comumente chamados de métodos modificadores porque geralmente alteram um valor. Métodos get são comumente chamados métodos acessores ou métodos de consulta. Um método predicado testa se uma condição é verdadeira ou falsa.

Seção 8.8 Composição

- Uma classe pode ter referências a objetos de outras classes como membros. Isso é chamado composição e, às vezes, é referido como um relacionamento *tem um*.

Seção 8.9 Tipos `enum`

- Todos os tipos `enum` são tipos por referência. Um tipo `enum` é declarado com uma declaração `enum`, que é uma lista separada por vírgulas de constantes `enum`. A declaração pode incluir opcionalmente outros componentes das classes tradicionais, como construtores, campos e métodos.
- Constantes `enum` são implicitamente `final`, porque declaram constantes que não devem ser modificadas.
- Constantes `enum` são implicitamente `static`.
- Qualquer tentativa de criar um objeto de um tipo `enum` com um operador `new` resulta em um erro de compilação.
- Constantes `enum` podem ser utilizadas em qualquer lugar em que constantes podem ser usadas, como nos rótulos `case` das instruções `switch` e para controlar instruções `for` aprimoradas.
- Cada constante `enum` em uma declaração `enum` é opcionalmente seguida por argumentos que são passados para o construtor `enum`.
- Para cada `enum`, o compilador gera um método `static` chamado `values` que retorna um array das constantes do `enum` na ordem em que elas foram declaradas.
- O método `EnumSet` `static` `range` recebe as primeiras e últimas constantes `enum` em um intervalo e retorna um `EnumSet` que contém todas as constantes entre essas duas constantes, inclusive.

Seção 8.10 Coleta de lixo

- A Java Virtual Machine (JVM) realiza a coleta de lixo automaticamente para reivindicar a memória ocupada pelos objetos que não estão mais em uso. Quando não há mais referências a um objeto, ele é marcado para coleta de lixo. A memória desse objeto pode ser reivindicada quando a JVM executa seu coletor de lixo.

Seção 8.11 Membros da classe `static`

- Uma variável `static` representa informações por toda a classe, que são compartilhadas entre os objetos da classe.
- Variáveis `static` têm escopo de classe. Os membros `public` `static` de uma classe podem ser acessados por meio de uma referência a qualquer objeto da classe ou qualificando o nome de membro com o nome de classe e um ponto (`.`). O código cliente só pode acessar os membros da classe `static` de uma classe `private` por meio dos métodos da classe.
- Membros da classe `static` existem assim que a classe é carregada na memória.
- Um método declarado `static` não pode acessar as variáveis de instância e os métodos de instância da classe, porque um método `static` pode ser chamado mesmo quando nenhum objeto da classe foi instanciado.
- A referência `this` não pode ser utilizada em um método `static`.

Seção 8.12 Importação `static`

- Uma declaração de importação `static` permite referenciar membros `static` importados sem o nome de classe e um ponto (`.`). Uma única declaração de importação `static` importa um membro `static` e uma importação `static` por demanda importa todos os membros `static` de uma classe.

Seção 8.13 Variáveis de instância `final`

- No contexto de um aplicativo, o princípio do menor privilégio afirma que deve ser concedida ao código somente a quantidade de privilégio e acesso que ele precisa para realizar sua tarefa designada.
- A palavra-chave `final` especifica que uma variável não é modificável. Essas variáveis devem ser inicializadas quando são declaradas ou por cada um dos construtores de uma classe.

Seção 8.14 Acesso de pacote

- Se nenhum modificador de acesso for especificado para um método ou variável quando esse método ou variável é declarado em uma classe, o método ou variável é considerado como tendo acesso de pacote.

Seção 8.15 Usando `BigDecimal` para cálculos monetários precisos

- Qualquer aplicativo que requer cálculos precisos de número de ponto flutuante sem erros de arredondamento — como aqueles em aplicações financeiras — deve usar a classe `BigDecimal` (pacote `java.math`).

- O método `BigDecimal` `static` `valueOf` com um argumento `double` retorna um `BigDecimal` que representa o valor exato especificado.
- O método `BigDecimal` `add` adiciona o argumento `BigDecimal` ao `BigDecimal` em que o método é chamado e retorna o resultado.
- `BigDecimal` fornece as constantes `ONE` (`1`), `ZERO` (`0`) e `TEN` (`10`).
- O método `BigDecimal` `pow` levanta seu primeiro argumento à potência especificada em seu segundo argumento.
- O método `BigDecimal` `multiply` multiplica o argumento `BigDecimal` pelo `BigDecimal` em que o método é chamado e retorna o resultado.
- A classe `NumberFormat` (pacote `java.text`) fornece as capacidades para formatar valores numéricos como `String`s específicas de localidade. O método `getCurrencyInstance` da classe `static` retorna um `NumberFormat` pré-configurado para valores de moedas específicos da localidade. O método `NumberFormat` realiza a formatação.
- Formatação específica da localidade é uma parte importante da internacionalização — o processo de personalização dos seus aplicativos para várias localidades e idiomas falados dos usuários.
- `BigDecimal` permite controlar como os valores são arredondados — por padrão, todos os cálculos são exatos e nenhum arredondamento ocorre. Se você não especifica como arredondar valores `BigDecimal` e um determinado valor não pode ser representado exatamente ocorre uma `ArithmeticException`.
- Você pode especificar o modo de arredondamento para `BigDecimal` fornecendo um objeto `MathContext` (pacote `java.math`) para o construtor da classe `BigDecimal` ao criar um `BigDecimal`. Você também pode fornecer um `MathContext` para vários métodos `BigDecimal` que realizam os cálculos. Por padrão, cada `MathContext` pré-configurado usa o assim chamado "arredondamento contábil".
- O escalonamento de um `BigDecimal` é o número de dígitos à direita do ponto decimal. Se você precisa de um `BigDecimal` arredondado para um dígito específico, chame o método `BigDecimal` `setScale`.

Exercício de revisão

8.1 Preencha as lacunas em cada uma das seguintes afirmações:

a) Um(a) _____ importa todos os membros `static` de uma classe.

b) O método `static` da classe `String` _____ é semelhante ao método `System.out.printf`, mas retorna uma `String` formatada em vez de exibir uma `String` em uma janela de comando.

c) Se um método contiver uma variável local com o mesmo nome de um dos campos da sua classe, a variável local _____ o campo no escopo desse método.

d) Os métodos `public` de uma classe também são conhecidos como _____ ou _____ da classe.

e) Uma declaração de _____ especifica uma classe a ser importada.

f) Se uma classe declarar construtores, o compilador não criará um(a) _____.

g) O método _____ de um objeto é chamado implicitamente quando um objeto aparece no código em que uma `String` é necessária.

h) Métodos *get* são comumente chamados de _____ ou _____.

i) Um método _____ testa se uma condição é verdadeira ou falsa.

j) Para cada `enum`, o compilador gera um método `static` chamado _____, que retorna um array das constantes do `enum` na ordem em que elas foram declaradas.

k) A composição às vezes é referida como um relacionamento _____.

l) Uma declaração de _____contém uma lista separada por vírgulas de constantes.

m) Uma variável _____ representa as informações de escopo de classe que são compartilhadas por todos os objetos da classe.

n) Uma declaração _____ importa um membro `static`.

o) O _____ declara que só deve ser concedida ao código a quantidade de privilégio e acesso que ele precisa para realizar sua tarefa designada.

p) A palavra-chave _____ especifica que uma variável não é modificável depois da inicialização em uma declaração ou em um construtor.

q) Uma declaração _____ importa somente as classes que o programa utiliza em um pacote em particular.

r) Métodos *set* são comumente chamados _____ porque eles geralmente alteram um valor.

s) Use a classe _____ para realizar cálculos monetários precisos.

t) Use a instrução _____ para indicar que ocorreu um problema.

Respostas do exercício de revisão

8.1 a) importação `static` sob demanda. b) `format`. c) espelha. d) serviços `public`, interface `public`. e) importação de tipo único. f) construtor padrão. g) `toString`. h) métodos acessores, métodos de consulta. i) predicado. j) `values`. k) *tem um*. l) `enum`. m) `static`. n) importação `static` única. o) princípio do menor privilégio. p) `final`. q) sob demanda. r) métodos modificadores. s) `BigDecimal`. t) `throw`.

Questões

8.2 (***Com base na Seção 8.14***) Explique a noção de acesso a pacotes no Java. Explique os aspectos negativos do acesso de pacote.

8.3 O que acontece quando um tipo de retorno, mesmo `void`, é especificado para um construtor?

8.4 (***Classe `Rectangle`***) Crie uma classe `Rectangle` com os atributos `length` e `width`, cada um dos quais assume o padrão de 1. Forneça os métodos que calculam o perímetro e a área do retângulo. A classe tem métodos *set* e *get* para o comprimento (`length`) e a largura (`width`). Os métodos *set* devem verificar se `length` e `width` são, cada um, números de ponto flutuante maiores que 0,0 e menores que 20,0. Escreva um programa para testar a classe `Rectangle`.

8.5 (***Modificando a representação interna de dados de uma classe***) Seria perfeitamente razoável que a classe `Time2` da Figura 8.5 represente a data/hora internamente como o número de segundos a partir da meia-noite em vez dos três valores inteiros `hour`, `minute` e `second`. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados. Modifique a classe `Time2` da Figura 8.5 para implementar `Time2` como o número de segundos desde a meia-noite e mostrar que não há alteração visível para os clientes da classe.

8.6 (***Classe `Savings Account`***) Crie uma classe `SavingsAccount`. Utilize uma variável `static` `annualInterestRate` para armazenar a taxa de juros anual para todos os correntistas. Cada objeto da classe contém uma variável de instância `private` `savingsBalance` para indicar a quantidade que o poupador atualmente tem em depósito. Forneça o método `calculateMonthlyInterest` para calcular os juros mensais multiplicando o `savingsBalance` por `annualInterestRate` dividido por 12 — esses juros devem ser adicionados ao `savingsBalance`. Forneça um método `static` `modifyInterestRate` que configure o `annualInterestRate` com um novo valor. Escreva um programa para testar a classe `SavingsAccount`. Instancie dois objetos `savingsAccount`, `saver1` e `saver2`, com saldos de R$ 2.000,00 e R$ 3.000,00, respectivamente. Configure `annualInterestRate` como 4% e então calcule o juro mensal de cada um dos 12 meses e imprima os novos saldos para os dois poupadores. Em seguida, configure `annualInterestRate` para 5%, calcule a taxa do próximo mês e imprima os novos saldos para os dois poupadores.

8.7 (***Aprimorando a classe `Time2`***) Modifique a classe `Time2` da Figura 8.5 para incluir um método `tick` que incrementa a data/hora armazenada em um objeto `Time2` em um segundo. Forneça um método `incrementMinute` para incrementar o minuto por um e o método `incrementHour` para incrementar a hora por uma. Escreva um programa que testa o método `tick`, o método `incrementMinute` e o método `incrementHour` para assegurar que eles funcionam corretamente. Certifique-se de testar os seguintes casos:

a) incrementar para o próximo minuto,

b) incrementar para a próxima hora e

c) incrementar para o próximo dia (isto é, 11:59:59 PM para 12:00:00 AM).

8.8 (***Aprimorando a classe `Date`***) Modifique a classe `Date` da Figura 8.7 para realizar uma verificação de erros nos valores inicializadores das variáveis de instância `month`, `day` e `year` (atualmente ela valida somente o mês e dia). Forneça um método `nextDay` para incrementar o dia por um. Escreva um programa que testa o método `nextDay` em um loop que imprime a data durante cada iteração para ilustrar que o método funciona corretamente. Teste os seguintes casos:

a) incrementar para o próximo mês e

b) incrementar para o próximo ano.

8.9 Reescreva o código na Figura 8.14 para utilizar uma declaração de importação separada para cada membro `static` da classe `Math` que é utilizado no exemplo.

8.10 Escreva um tipo `enum` `TrafficLight`, cuja constante (`RED`, `GREEN`, `YELLOW`) aceite um parâmetro — a duração da luz. Escreva um programa para testar o `enum` `TrafficLight` de modo que ele exiba a constante `enum` e suas durações.

8.11 (***Números complexos***) Crie uma classe chamada `Complex` para realizar aritmética com números complexos. Os números complexos têm a forma

parteReal + *parteImaginária* * *i*

onde *i* é

$$\sqrt{-1}$$

Escreva um programa para testar sua classe. Utilize variáveis de ponto flutuante para representar os dados `private` da classe. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. Forneça um construtor sem argumento com valores padrão caso nenhum inicializador seja fornecido. Forneça métodos `public` que realizam as seguintes operações:

a) Somar dois números `Complex`: as partes reais são somadas de um lado e as partes imaginárias, de outro.

b) Subtrair dois números `Complex`: a parte real do operando direito é subtraída da parte real do operando esquerdo e a parte imaginária do operando direito é subtraída da parte imaginária do operando esquerdo.

c) Imprima números `Complex` na forma (*parteReal*, *parteImaginária*).

8.12 (***Classe `DateAndTime`***) Crie uma classe `DateAndTime` que combina a classe `Time2` modificada do Exercício 8.7 e a classe `Date` modificada do Exercício 8.8. Modifique o método `incrementHour` para chamar o método `nextDay` se a data/hora for incrementada para o

próximo dia. Modifique métodos `toString` e `toUniversalString` para gerar uma saída da data além da hora. Escreva um programa para testar a nova classe `DateAndTime`. Especificamente, teste o incremento de tempo para o próximo dia.

8.13 **(*Conjunto de inteiros*)** Crie a classe `IntegerSet`. Cada objeto `IntegerSet` pode armazenar inteiros no intervalo de 0 a 100. O conjunto é representado por um array de `boolean`s. O elemento do array `a[i]` é `true` se o inteiro *i* estiver no conjunto. O elemento do array `a[j]` é `false` se o inteiro *j* não estiver no conjunto. O construtor sem argumento inicializa o array como um "conjunto vazio" (isto é, todos os valores `false`).

Forneça os seguintes métodos: o método `static union` cria um conjunto que é a união teórica de dois conjuntos existentes (isto é, um elemento do array do novo conjunto é configurado como `true` se esse elemento for `true` em qualquer um dos conjuntos existentes ou em ambos — caso contrário, o elemento do novo conjunto é configurado como `false`). O método `static intersection` cria um conjunto que é a interseção teórica de dois conjuntos existentes (isto é, um elemento do array do novo conjunto é configurado como `false` se esse elemento for `false` em qualquer um ou em ambos os conjuntos existentes — caso contrário, o elemento do novo conjunto é configurado como `true`). O método `insertElement` insere um novo inteiro *k* em um conjunto (configurando `a[k]` como `true`). O método `deleteElement` exclui o inteiro *m* (configurando `a[m]` como `false`). O método `toString` retorna uma `String` contendo um conjunto como uma lista de números separados por espaços. Inclua somente os elementos que estão presentes no conjunto. Utilize `---` para representar um conjunto vazio. O método `isEqualTo` determina se dois conjuntos são iguais. Escreva um programa para testar a classe `IntegerSet`. Instancie vários objetos `IntegerSet`. Teste se todos os seus métodos funcionam adequadamente.

8.14 **(*Classe `Data`*)** Crie uma classe `Date` com as seguintes capacidades:

a) Gerar saída da data em múltiplos formatos, como

```
MM/DD/YYYY
June 14, 1992
DDD YYYY
```

b) Utilizar construtores sobrecarregados para criar objetos `Date` inicializados com datas dos formatos na parte (a). No primeiro caso, o construtor deve receber três valores inteiros. No segundo caso, deve receber uma `String` e dois valores inteiros. No terceiro caso, deve receber dois valores inteiros, o primeiro representando o número de dias no ano. [*Dica:* para converter a representação de `String` do mês em um valor numérico, compare as `Strings` utilizando o método `equals`. Por exemplo, se `s1` e `s2` forem `strings`, a chamada de método `s1.equals(s2)` retornará `true` se as `strings` forem idênticas, caso contrário retornará `false`.]

8.15 **(*Números racionais*)** Crie uma classe chamada `Rational` para realizar aritmética com frações. Escreva um programa para testar sua classe. Use variáveis de inteiros para representar as variáveis de instância `private` da classe — o `numerator` e o `denominator`. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve armazenar a fração em uma forma reduzida. A fração

```
2/4
```

é equivalente a `1/2` e seria armazenada no objeto como 1 no `numerator` e 2 no `denominator`. Forneça um construtor sem argumento com valores padrão caso nenhum inicializador seja fornecido. Forneça métodos `public` que realizam cada uma das operações a seguir:

a) Somar dois números `Rational`: o resultado da adição deve ser armazenado na forma reduzida. Implemente isso como um método `static`.

b) Subtrair dois números `Rational`: o resultado da subtração deve ser armazenado na forma reduzida. Implemente isso como um método `static`.

c) Multiplicar dois números `Rational`: o resultado da multiplicação deve ser armazenado na forma reduzida. Implemente isso como um método `static`.

d) Dividir dois números `Rational`: o resultado da divisão deve ser armazenado na forma reduzida. Implemente isso como um método `static`.

e) Retorne uma representação `String` de um número `Rational` na forma `a/b`, onde `a` é o `numerator` e `b` é o `denominator`.

f) Retorne uma representação `String` de um número `Rational` no formato de ponto flutuante. (Considere a possibilidade de fornecer capacidades de formatação que permitam que o usuário da classe especifique o número de dígitos de precisão à direita do ponto de fração decimal.)

8.16 **(*Classe `Huge Integer`*)** Crie uma classe `HugeInteger` que utiliza um array de 40 elementos de dígitos para armazenar inteiros com até 40 dígitos. Forneça os métodos `parse`, `toString`, `add` e `subtract`. O método `parse` deve receber uma `String`, extrair cada dígito usando o método `charAt` e colocar o valor inteiro equivalente de cada dígito no array de inteiros. Para comparar objetos `HugeInteger`, forneça os métodos a seguir: `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` e `isLessThanOrEqualTo`. Cada um destes é um método predicado que retorna `true` se o relacionamento estiver contido entre os dois objetos `HugeInteger` e retorna `false` se o relacionamento não estiver contido. Forneça um método predicado `isZero`. Se você se sentir ambicioso, forneça também os métodos `multiply`, `divide` e `remainder`. [*Observação:* valores `boolean` primitivos podem ser gerados como as palavras "true" ou "false" com o especificador de formato `%b`.]

8.17 **(*Jogo da velha*)** Crie uma classe `TicTacToe` que permitirá escrever um programa para reproduzir o jogo da velha. A classe contém um array bidimensional privado 3 por 3. Use um tipo `enum` para representar o valor em cada célula do array. As constantes `enum` devem ser nomeadas `X`, `O` e `EMPTY` (para uma posição que não contém `X` ou `O`). O construtor deve inicializar os elementos do tabuleiro para `EMPTY`. Permita dois jogadores humanos. Para onde quer que o primeiro jogador se mova, coloque um `X` no quadrado especificado; coloque um `O` no local para o qual o segundo jogador se mover. Todo movimento deve ocorrer em um quadrado vazio. Depois de cada jogada, determine se o jogo foi ganho e se aconteceu um empate. Se você se sentir motivado, modifique seu programa de modo que o computador faça o

movimento para um dos jogadores. Além disso, permita que o jogador especifique se quer ser o primeiro ou o segundo. Se você se sentir excepcionalmente motivado, desenvolva um programa que jogue o Tic-Tac-Toe tridimensional em uma grade 4 por 4 por 4. [*Observação:* isso é um projeto extremamente desafiador!]

8.18 (***Classe `Account` com saldo `BigDecimal`***) Reescreva a classe `Account` da Seção 3.5 para armazenar o `balance` como um objeto `BigDecimal` e para realizar todos os cálculos usando `BigDecimals`.

Fazendo a diferença

8.19 (***Projeto: classe de resposta a emergência***) O serviço de resposta de emergência norte-americano, 9-1-1, conecta os autores da chamada a um serviço de resposta de serviço público (Public Service Answering Point, PSAP) *local*. Tradicionalmente, o PSAP solicitaria ao chamador informações de identificação — incluindo o endereço, número de telefone e a natureza da emergência do autor da chamada, então enviaria os socorristas de emergência apropriados (como a polícia, uma ambulância ou o corpo de bombeiros). O *Enhanced 9-1-1 (ou E*9-1-1*)* usa computadores e bancos de dados para determinar o endereço físico do autor da chamada, direciona a chamada para o PSAP mais próximo e exibe o número de telefone e o endereço do autor da chamada para quem a recebe. O *Wireless Enhanced 9-1-1* fornece a quem recebe a chamada informações de identificação para chamadas sem fio. Implementado em duas fases, a primeira exigiu que operadoras fornecessem o número de telefone sem fio e a localização do local do celular ou estação base que transmite a chamada. A segunda exigiu que as operadoras fornecessem a localização do autor da chamada (utilizando tecnologias como GPS). Para saber mais sobre 9-1-1, visite `http://www.fcc.gov/pshs/services/911-services/Welcome.html` e `http://people.howstuffworks.com/9-1-1.htm`.

Uma parte importante da criação de uma classe é determinar os atributos dela (variáveis de instância). Para este exercício de design de classe, pesquise serviços 9-1-1 na internet. Então, crie uma classe chamada `Emergency` que pode ser usada em um sistema de resposta de emergência 9-1-1 orientado a objetos. Liste os atributos que um objeto dessa classe pode usar para representar a emergência. Por exemplo, a classe pode incluir informações sobre quem relatou a emergência (incluindo o número de telefone), o local da emergência, a data/hora do relatório, a natureza da emergência, o tipo e o status da resposta. Os atributos da classe devem descrever completamente a natureza do problema e o que acontece para resolvê-lo.