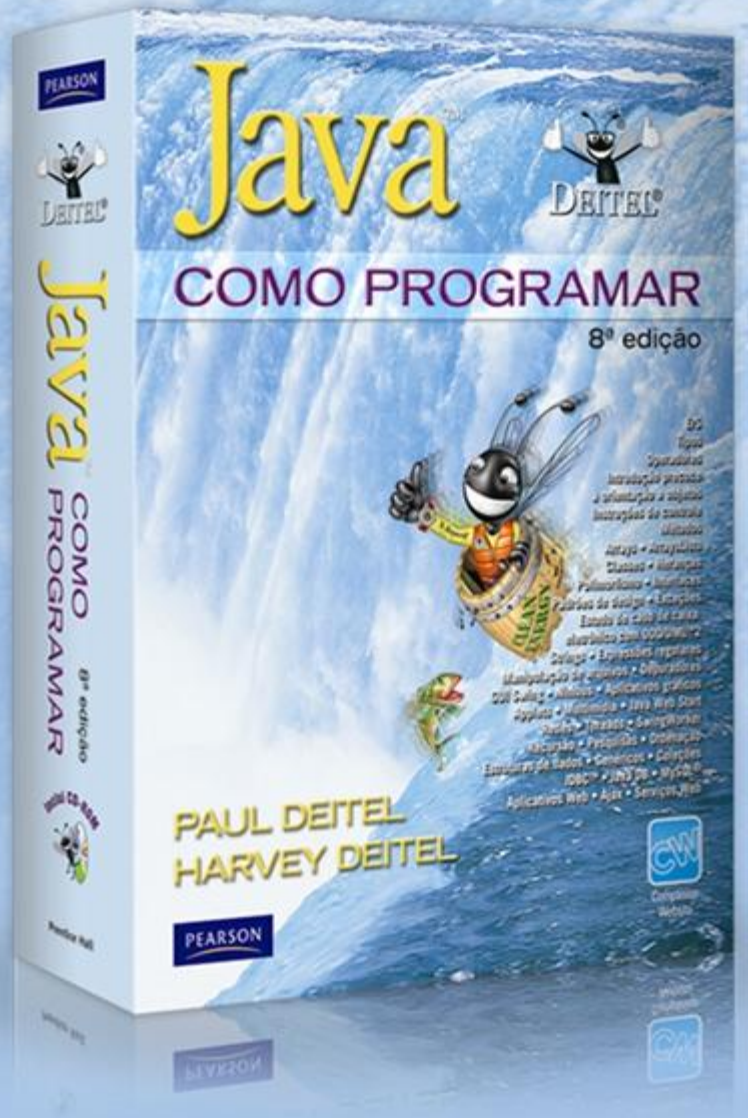
Capítulo 8
Classes e objetos: uma visão mais aprofundada



Java™ Como Programar, 8/E

OBJETIVOS

Neste capítulo, você aprenderá:

- O encapsulamento e o ocultamento de dados.
- A utilizar a palavra-chave `this`.
- A utilizar variáveis e métodos `static`.
- A importar membros `static` de uma classe.
- A utilizar o tipo `enum` para criar conjuntos de constantes com identificadores únicos.
- A declarar constantes `enum` com parâmetros.
- A organizar classes em pacotes para promover a reutilização.

8.1 Introdução

- Análise mais profunda da construção de classes, controle de acesso a membros de uma classe e criação de construtores.
- Composição — uma capacidade que permite a uma classe conter referências a objetos de outras classes como membros.
- Mais detalhe sobre tipos `enum`.
- O capítulo também discute os membros de classe `static` e variáveis de instância `final` em detalhes.
- Mostra como organizar classes em pacotes para ajudar a gerenciar grandes aplicativos e promove a reutilização.

8.2 Estudo de caso da classe `Time`

- A classe `Time1` representa a hora do dia.
- Variáveis de instância `private int hour`, `minute` e `second` representam a hora no formato hora universal (formato de relógio de 24 horas em que as horas estão no intervalo de 0 a 23).
- Métodos `public setTime`, `toUniversalString` e `toString`.

  Chamados de **serviços public** ou **interface public** que a classe fornece a seus clientes.


```
1  // Figura 8.1: Time1.java
2  // Declaração de classe Time1 mantém a 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 hora usando formato universal;
11    // assegura que os dados permaneçam consistentes configurando valores inválidos
11    // como zero
12    public void setTime( int h, int m, int s )
13    {
14       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // valida horas
15       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // valida minutos
16       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida segundos
17    } // fim do método setTime
18
19    // converte em String no formato de hora universal (HH:MM:SS)
20    public String toUniversalString()
21    {
22       return String.format( "%02d:%02d:%02d", hour, minute, second );
23    } // fim do método do toUniversalString
24
```

Variáveis de instância representam a hora no formato de 24 horas

Valida os valores iniciais de hora, minuto e segundo

Formato de hora de 24 horas

Figura 8.1 | Declaração da classe `Time1` mantém a hora no formato de 24 horas. (Parte 1 de 2.)


```
25       // converte em String no formato padrão hora (H:MM:SS AM ou PM)
26       public String toString()
27       {
28          return String.format( "%d:%02d:%02d %s",
29             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
30             minute, second, ( hour < 12 ? "AM" : "PM" ) );
31       } // fim do método toString
32    } // fim da classe Time1
```

Formata a hora no formato de hora de 12 horas; esse também é o formato `String` padrão para `Time1`

Figura 8.1 | Declaração da classe `Time1` mantém a hora no formato de 24 horas. (Parte 2 de 2.)


```
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 hora
12       System.out.print( "The initial universal time is: " );
13       System.out.println(time.toUniversalString() );
14       System.out.print( "The initial standard time is: " );
15       System.out.println(time.toString() );
16       System.out.println(); // gera saída de uma linha em branco
17
18       // altera a hora e gera saída da hora atualizada
19       time.setTime( 13, 27, 6 );
20       System.out.print( "Universal time after setTime is: " );
21       System.out.println(time.toUniversalString() );
22       System.out.print( "Standard time after setTime is: " );
23       System.out.println(time.toString() );
24       System.out.println(); // gera saída de uma linha em branco
```

Cria o objeto `Time1` padrão

Obtém a representação `String` da hora no formato de 24 horas

Obtém a representação `String` da hora no formato de 12 horas

Configura a hora usando valores válidos para hora, minuto e segundo

Figura 8.2 | Objeto `Time1` utilizado em um aplicativo. (Parte 1 de 2.)

```
25
26          // configura a hora com valores inválidos; gera saída da hora atualizada
27          time.setTime( 99, 99, 99 );
28          System.out.println( "After attempting invalid settings:" );
29          System.out.print( "Universal time: " );
30          System.out.println(time.toUniversalString() );
31          System.out.print( "Standard time: " );
32          System.out.println(time.toString() );
33       } // fim de main
34    } // fim da classe Time1Test
```

Configura a hora usando valores inválidos para hora, minuto e segundo

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Figura 8.2 | Objeto `Time1` utilizado em um aplicativo. (Parte 2 de 2.)

- A classe `Time1` não declara um construtor, portanto a classe tem um construtor padrão fornecido pelo compilador.
- Cada variável de instância recebe implicitamente o valor padrão de `0` para um `int`.
- Variáveis de instância também podem ser inicializadas quando declaradas no corpo da classe utilizando a mesma sintaxe de inicialização de uma variável local.

- Um objeto `Time1` sempre contém *dados consistentes.*

  Os valores dos dados do objeto sempre são mantidos em um intervalo, mesmo se os valores fornecidos como argumentos para o método `setTime` estiverem *incorretos.*
- Neste exemplo, zero é um valor *consistente* para `hour`, `minute` e `second`.
- `hour`, `minute` e `second` são todos configurados como zero por padrão; assim, um objeto `Time1` contém dados consistentes a partir do momento em que é criado.

- É importante distinguir entre um *valor correto* e um *valor consistente*.

  Um valor consistente para `minute` deve estar no intervalo entre 0 e 59.

  Um valor correto para `minute` determinado aplicativo seria o minuto real naquela hora do dia.

  Configurando a hora em um relógio.

  Se a hora real estiver 17 minutos depois da hora e você acertar o relógio acidentalmente como 19 minutos depois da hora, 19 é um valor *consistente* (0 a 59), mas não um *valor correto*.

  Se você acertar o relógio para 17 minutos depois da hora, então 17 é um valor correto — e um valor correto é *sempre* um valor consistente.

- Para valores inconsistentes, podemos simplesmente deixar o objeto no estado atual, sem alterar a variável de instância.

  Objetos `Time` iniciam em um estado consistente e o método `setTime` rejeita qualquer valor inconsistente.

  Os objetos estão sempre *garantidamente* em um estado consistente.

  Muitas vezes esse seria o último estado correto do objeto, o qual alguns designers acham ser superior a configurar as variáveis de instância como zero.

- Problemas potenciais

  As abordagens discutidas até aqui não informam o código do cliente sobre valores inconsistentes.

  `setTime` poderia retornar um valor tal como `true` se todos os valores fossem consistentes e `false` se todos os valores fossem inconsistentes.

  O chamador verificaria o valor de retorno, e se ele fosse `false`, tentaria estabelecer a hora novamente.

  Problema: Algumas tecnologias Java (como JavaBeans) exigem que os métodos *set* retornem `void`.

  No Capítulo 11, "Tratamento de exceções", você aprenderá técnicas que permitem que seus métodos indiquem quando valores inconsistentes são recebidos.

- As variáveis de instância `hour`, `minute` e `second` são, todas elas, declaradas `private`.
- A representação real dos dados utilizada dentro da classe não diz respeito aos clientes da classe.
- Seria perfeitamente razoável `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.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.1

As classes simplificam a programação, porque o cliente pode utilizar somente os métodos `public` *expostos pela 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. Os clientes geralmente se preocupam com* o que *a classe faz, mas não* como *a classe faz isso.*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.2

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.

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 Capítulo 9 introduz o modificador de acesso `protected`.
- Métodos public apresentam aos clientes da classe uma visualização dos serviços que a classe fornece (a interface `public` da classe).
- Os clientes não precisam se preocupar com a maneira como a classe realiza suas tarefas.

  Por essa razão, as variáveis `private` e os métodos `private` da classe (isto é, os detalhes da sua implementação) não são acessíveis aos seus clientes.
- Membros `private` de uma classe não são acessíveis fora da classe.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.1

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


```
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    } // fim de main
13 } // fim da classe MemberAccessTest
```

Cada uma dessas instruções tenta acessar dados que são `private` para `Time1`

Figura 8.3 | Membros privados da classe `Time1` não são acessíveis. (Parte 1 de 2.)

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

Figura 8.3 | Membros privados da classe `Time1` não são acessíveis. (Parte 2 de 2.)

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

- Qualquer objeto pode acessar uma referência para si mesmo com palavra-chave **this**.
- Quando um método não `static` 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 que o objeto deve ser manipulado.

  Também é possível usar a palavra-chave `this` explicitamente no corpo de um método não `static`.
- É possível usar a referência `this` implícita e explicitamente.

- 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.
- Quando um arquivo código-fonte `(.java)` contém várias declarações de classe, o compilador coloca ambos os arquivos de classe dessas classes no mesmo diretório.
- Um arquivo de código-fonte pode conter somente uma classe `public` — caso contrário, um erro de compilação ocorre.
- Classes não `public` só podem ser utilizadas por outras classes no mesmo pacote.

```
 1   // Figura 8.4: ThisTest.java
 2   // this usado implícita e explicitamente para referenciar 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      } // fim de main
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
```

Figura 8.4 | `this` usado implícita e explicitamente como uma referência a membros de um objeto. (Parte 1 de 3)


```
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 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 os minutos do objeto "this"
27         this.second = second; // configura os segundos do objeto "this"
28      } // fim do construtor SimpleTime
29
30      // usa "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      } // fim do método buildString
37
```

A referência `this` permite acessar explicitamente variáveis de instância quando elas estão sombreadas pelas variáveis locais de mesmo nome

A referência `this` não é exigida para chamar outros métodos da mesma classe

Figura 8.4 | `this` usado implícita e explicitamente como uma referência a membros de um objeto. (Parte 1 de 3)


```
38       // converte em String no formato de 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       } // fim do método do toUniversalString
47    } // fim da classe SimpleTime
```

"this" não é requerido aqui uma vez que as variáveis de instância não estão sombreadas

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

Figura 8.4 | this usado implícita e explicitamente como uma referência a membros de um objeto. (Parte 3 de 3)

- `SimpleTime` declara três variáveis de instância `private` — `hour`, `minute` e `second`.
- Se os nomes de parâmetro para o construtor forem idênticos aos nomes das variáveis de instância da classe
  - Não recomendamos essa prática.
  - Utilize-o aqui para "sombrear" (ocultar) a instância correspondente.
  - Ilustra um caso em que uso explícito da referência `this` é exigido.
- Se um método contiver uma variável local com o mesmo nome de um campo, esse método usa a variável local em vez do campo.
  - Nesse caso, a variável local "sombreia" o campo no escopo do método.
- O método pode utilizar a referência `this` para referenciar o campo sombreado explicitamente.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.2

Frequentemente é um erro de lógica quando um método contém um parâmetro ou variável local com o mesmo nome de um campo da classe. Nesse caso, utilize a referência `this` *se desejar acessar o campo da classe — caso contrário, o parâmetro ou variável local do método será referenciado.*

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 8.1

Evite nomes de parâmetros ou variáveis locais nos métodos que conflitem com nomes de campos. Isso ajuda a evitar bugs sutis, difíceis de corrigir.

Java™ COMO PROGRAMAR 8ª edição

Dica de desempenho 8.1

O Java economiza espaço de 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 (isto é, campos não `static`*). Cada método da classe utiliza implicitamente* `this` *para determinar o objeto específico da classe a manipular.*

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

- **Construtores sobrecarregados** permitem que objetos de uma classe sejam inicializados de diferentes maneiras.
- Para sobrecarregar construtores, simplesmente forneça múltiplas declarações de construtor com assinaturas diferentes.
- Lembre-se de que o compilador diferencia assinaturas pelo *número* de parâmetros, os *tipos de parâmetro* e a *ordem* dos tipos de parâmetro em cada assinatura.

- A classe `Time2` (Figura 8.5) contém cinco construtores sobrecarregados que fornecem maneiras convenientes de inicializar os objetos da nova classe `Time2`.
- O compilador invoca o construtor apropriado correspondendo o número, os tipos e a ordem dos tipos dos argumentos especificados na chamada de construtor com o número, os tipos e a ordem dos tipos dos parâmetros especificados em cada declaração de construtor.


```
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 : inicializa cada variável de instância
11    // com zero; assegura que objetos Time2 iniciam em um estado consistente
12    public Time2()
13    {
14       this( 0, 0, 0 ); // invoca o construtor Time2 com três argumentos
15    } // fim do construtor sem argumento Time2
16
17    // Construtor Time2: hour fornecido, min e sec padronizados como 0
18    public Time2( int h )
19    {
20       this( h, 0, 0 ); // invoca o construtor Time2 com três argumentos
21    } // fim do construtor de um argumento Time2
22
```

Invoca o construtor de três argumentos

Invoca o construtor de três argumentos

Figura 8.5 | Classe `Time2` com construtores sobrecarregados. (Parte 1 de 5.)

```
23      // Construtor Time2: hour e min fornecidos, sec padronizado como 0
24      public Time2( int h, int m )
25      {
26         this( h, m, 0 ); // invoca o construtor Time2 com três argumentos
27      } // fim do construtor de dois argumentos Time2
28
29      // Construtor Time2: hour, min e sec fornecidos
30      public Time2( int h, int m, int s )
31      {
32         setTime( h, m, s ); // invoca setTime para validar a hora
33      } // fim do construtor de três argumentos Time2
34
35      // Construtor Time2: outro objeto Time2 fornecido
36      public Time2( Time2 time )
37      {
38         // invoca o construtor de três argumentos Time2
39         this( time.getHour(), time.getMinute(), time.getSecond() );
40      } // fim do construtor Time2 com um argumento de objeto Time2
41
```

Invoca o construtor de três argumentos

Invoca `setTime` para validar os dados

Invoca o construtor de três argumentos

Figura 8.5 | Classe `Time2` com construtores sobrecarregados. (Parte 2 de 5.)

```
42       // Métodos set
43       // configura um novo valor de hora usando o formato universal;
44       // assegura que os dados permaneçam consistentes configurando
44       // valores inválidos como zero
45       public void setTime( int h, int m, int s )
46       {
47          setHour( h ); // configura hour
48          setMinute( m ); // configura minute
49          setSecond( s ); // configura second
50       } // fim do método setTime
51
52       // valida e configura a hora
53       public void setHour( int h )
54       {
55          hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
56       } // fim do método setHour
57
58       // valida e configura os minutos
59       public void setMinute( int m )
60       {
61          minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62       } // fim do método setMinute
63
```

Figura 8.5 | Classe `Time2` com construtores sobrecarregados. (Parte 3 de 5.)

```
64       // valida e configura os segundos
65       public void setSecond( int s )
66       {
67          second = ( ( s >= 0 && s < 60 ) ? s : 0 );
68       } // fim do método setSecond
69
70       // Métodos get
71       // obtém valor da hora
72       public int getHour()
73       {
74          return hour;
75       } // fim do método getHour
76
77       // obtém valor dos minutos
78       public int getMinute()
79       {
80          return minute;
81       } // fim do método getMinute
82
83       // obtém valor dos segundos
84       public int getSecond()
85       {
86          return second;
87       } // fim do método getSecond
```

Figura 8.5 | Classe `Time2` com construtores sobrecarregados. (Parte 4 de 5.)

```
88
89       // converte em String no formato de hora universal (HH:MM:SS)
90       public String toUniversalString()
91       {
92          return String.format(
93             "%02d:%02d:%02d", getHour(), getMinute(), getSecond() );
94       } // fim do método do toUniversalString
95
96       // converte em String no formato padrão de data (H:MM:SS AM ou PM)
97       public String toString()
98       {
99          return String.format( "%d:%02d:%02d %s",
100            ( (getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12 ),
101            getMinute(), getSecond(), ( getHour() < 12 ? "AM" : "PM" ) );
102      } // fim do método toString
103   } // fim da classe Time2
```

Figura 8.5 | Classe `Time2` com construtores sobrecarregados. (Parte 5 de 5.)


```
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( 27, 74, 99 ); // 00:00:00
13       Time2 t6 = new Time2( t4 ); // 12:25:42
14
15       System.out.println( "Constructed with:" );
16       System.out.println( "t1: all arguments defaulted" );
17       System.out.printf( "   %s\n", t1.toUniversalString() );
18       System.out.printf( "   %s\n", t1.toString() );
19
20       System.out.println(
21          "t2: hour specified; minute and second defaulted" );
22       System.out.printf( "   %s\n", t2.toUniversalString() );
23       System.out.printf( "   %s\n", t2.toString() );
24
```

O compilador determina qual construtor chamar com base no número e nos tipos dos argumentos

Figura 8.6 | Construtores sobrecarregados utilizados para inicializar objetos `Time2`. (Parte 1 de 3.)

```
25          System.out.println(
26             "t3: hour and minute specified; second defaulted" );
27          System.out.printf( "   %s\n", t3.toUniversalString() );
28          System.out.printf( "   %s\n", t3.toString() );
29
30          System.out.println( "t4: hour, minute and second specified" );
31          System.out.printf( "   %s\n", t4.toUniversalString() );
32          System.out.printf( "   %s\n", t4.toString() );
33
34          System.out.println( "t5: all invalid values specified" );
35          System.out.printf( "   %s\n", t5.toUniversalString() );
36          System.out.printf( "   %s\n", t5.toString() );
37
38          System.out.println( "t6: Time2 object t4 specified" );
39          System.out.printf( "   %s\n", t6.toUniversalString() );
40          System.out.printf( "   %s\n", t6.toString() );
41       } // fim de main
42    } // fim da classe Time2Test
```

Figura 8.6 | Construtores sobrecarregados utilizados para inicializar objetos `Time2`. (Parte 2 de 3.)

```
t1: all arguments defaulted
   00:00:00
   12:00:00 AM
t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM
t3: hour and minute specified; second defaulted
   21:34:00
   9:34:00 PM
t4: hour, minute and second specified
   12:25:42
   12:25:42 PM
t5: all invalid values specified
   00:00:00
   12:00:00 AM
t6: Time2 object t4 specified
   12:25:42
   12:25:42 PM
```

Figura 8.6 | Construtores sobrecarregados utilizados para inicializar objetos `Time2`. (Parte 3 de 3.)

- Um programa pode declarar o chamado **construtor sem argumento** que é invocado sem argumentos.
- Esse construtor simplesmente inicializa o objeto como especificado no corpo do construtor.
- Usando `this` no corpo de um construtor para chamar outro construtor da mesma classe.

  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.
- Assim que você declara qualquer construtor em uma classe, o compilador não fornecerá um construtor padrão.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.3

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

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.4

Um construtor pode chamar métodos da classe. Esteja ciente de que as variáveis de instância talvez ainda não estejam em um estado consistente, porque o construtor está no processo de inicialização do objeto. Usar variáveis de instância antes de elas serem inicializadas adequadamente é um erro de lógica.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.3

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`*).*

- Notas sobre os métodos *set* and *get* e os construtores da classe `Time2`.
- Os métodos podem acessar dados privados da classe diretamente sem chamar os métodos *set* e *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 fizéssemos essa alteração, somente o corpo dos métodos que acessam os dados `private` diretamente precisaria mudar — em particular, os métodos *set* and *get* individuais para `hour`, `minute` e `second`.

  Não haveria necessidade de modificar o corpo dos métodos `setTime`, `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.
- De maneira semelhante, cada construtor `Time2` poderia ser escrito para incluir uma cópia das instruções apropriadas a partir dos métodos `setHour`, `setMinute` e `setSecond`.

  Fazer isso talvez seja um pouco mais eficiente, porque a chamada extra ao construtor e a chamada a `setTime` são eliminadas. Mas duplicar instruções em múltiplos métodos ou construtores dificulta a alteração da representação interna de dados da classe.

  Fazer com que os construtores `Time2` chamem o construtor com três argumentos (ou mesmo chamem `setTime` diretamente) requer que as alterações na implementação de `setTime` sejam feitas somente uma vez.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.4

Ao implementar um método de uma classe, utilize os métodos set *e* get *da classe para acessar os dados* `private` *da classe. Isso simplifica a manutenção do código e reduz a probabilidade de erros.*

8.6 Construtores padrão e sem argumentos

- Cada classe deve ter pelo menos um construtor.
- Se você não fornecer nenhum construtor na declaração de uma classe, o compilador cria um construtor padrão que não aceita argumentos ao ser 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).
- Se a sua classe declarar construtores, o compilador não criará um construtor padrão.

  Nesse caso, você deve declarar um construtor sem argumentos se a inicialização padrão for requerida.

  Como ocorre com um construtor padrão, um construtor sem argumentos é invocado com parênteses vazios.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.5

Um erro de compilação ocorre se um programa tentar inicializar um objeto de uma classe passando o número incorreto ou tipos de argumentos para o construtor da classe.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.5

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. O Java determina quais métodos são construtores localizando aqueles que têm o mesmo nome da classe e que não especificam um tipo de retorno.

8.7 Notas sobre os métodos *set* e *get*

- As classes costumam fornecer métodos `public` para permitir aos clientes configurar (*set* , isto é, atribuir valores a) ou obter (*set* , isto é, obter os valores de) variáveis de instância `private`.
- Os métodos *set* também são comumente chamados de métodos modificadores, porque em geral alteram o estado de um objeto — isto é, modificam os valores de variáveis de instância.
- Os métodos *get* também são comumente chamados de **métodos de acesso** ou **métodos de consulta.**

- Parece que fornecer as capacidades de *set* e *get* é essencialmente o mesmo que tornar `public` as variáveis de instância.

  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 `public` *get* certamente permitirá que outros métodos a acessem, mas o método *get* pode controlar como o cliente pode acessá-la.

  Um método `public` *set* pode — e deve — avaliar cuidadosamente as tentativas de modificar o valor da variável a fim de assegurar que o novo valor é consistente para esse item de dados.
- Embora métodos *set* e *get* forneça acesso a dados `private`, o acesso é restrito pela implementação dos métodos.

- ***Teste de validade em métodos set***
- Os benefícios da integridade de dados não seguem automaticamente simplesmente porque as variáveis de instância são declaradas `private` — você deve fornecer a verificação de validade.
- ***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**.

  Exemplo: O método `isEmpty` de `ArrayList`, que retorna `true` se o `ArrayList` estiver vazio.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.6

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 8.2

Os métodos set *e* get *ajudam a criar classes que são mais fáceis de depurar e manter. Se apenas um método realizar uma tarefa particular, como configurar a hora em um objeto* `Time2`*, é mais fácil depurar e manter a classe. Se a* `hour` *não for configurada adequadamente, o código que na verdade modifica a variável de instância* `hour` *estará localizado no corpo do método — apenas* `setHour`*. Portanto, seus esforços de depuração podem se concentrar no método* `setHour`*.*

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***.
- Exemplo: Um objeto `AlarmClock` precisa saber a hora atual e a hora em que o alarme deve tocar, portanto, é razoável incluir duas referências a objetos `Time` em um objeto `AlarmClock`.

```
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    // construtor: chama checkMonth para confirmar o valor adequado para month;
11    // chama checkDay para confirmar o valor adequado para day
12    public Date( int theMonth, int theDay, int theYear )
13    {
14       month = checkMonth( theMonth ); // valida month
15       year = theYear; // poderia validar year
16       day = checkDay( theDay ); // valida day
17
18       System.out.printf(
19          "Date object constructor for date %s\n", this );
20    } // fim do construtor Date
21
```

Figura 8.7 | Declaração da classe `Date`. (Parte 1 de 3.)


```
22       // método utilitário para confirmar o valor adequado de month
23       private int checkMonth( int testMonth )
24       {
25          if ( testMonth > 0 && testMonth <= 12 ) // valida month
26             return testMonth;
27          else // month é inválido
28          {
29             System.out.printf(
30                "Invalid month (%d) set to 1.", testMonth );
31             return 1; // mantém objeto em estado consistente
32          } // fim de else
33       } // fim do método checkMonth
34
35       // método utilitário para confirmar o valor adequado de day com base em month e year
36       private int checkDay( int testDay )
37       {
38          int[] daysPerMonth =
39             { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
40
41          // verifica se day está no intervalo para month
42          if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
43             return testDay;
44
```

Figura 8.7 | Declaração da classe `data`. (Parte 2 de 3.)

```
45          // verifica ano bissexto
46          if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
47               ( year % 4 == 0 && year % 100 != 0 ) ) )
48             return testDay;
49
50          System.out.printf( "Invalid day (%d) set to 1.", testDay );
51          return 1;  // mantém objeto em estado consistente
52       } // fim do método checkDay
53
54       // retorna uma String no formato mês/dia/ano
55       public String toString()
56       {
57          return String.format( "%d/%d/%d", month, day, year );
58       } // fim do método toString
59    } // fim da classe Date
```

Figura 8.7 | Declaração da classe `data`. (Parte 3 de 3.)

```
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 name, dateOfBirth e dateOfHire
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // fim do construtor Employee
20
```

Referências a outros objetos compostos na classe `Employee`

Figura 8.8 | A classe `Employee` com referência a outros objetos. (Parte 1 de 2.)


```
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       } // fim do método toString
27    } // fim da classe Employee
```

Figura 8.8 | A classe `Employee` com referência a outros objetos. (Parte 2 de 2.)

```
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    } // fim de main
14 } // fim da classe EmployeeTest
```

Objetos `Date` usados para inicializar `Employee`

Obtém a representação `String` de `Employee` chamando `toString` implicitamente

```
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 | Demonstração de composição.

8.9 Enumerações

- O tipo `enum` básico define um conjunto de constantes representadas como identificadores únicos.
- Como as classes, 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.

- Cada declaração `enum` declara uma classe `enum` com as seguintes restrições:
  - 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 utilizadas, como nos rótulos `case` das instruções `switch` e para controlar instruções `for` aprimoradas.
  - Declarações `enum` contém duas partes — as constantes `enum` e os outros membros do tipo `enum`.
  - Um construtor `enum` pode especificar qualquer número de parâmetros e pode ser sobrecarregado.
- Para cada `enum`, o compilador gera os **valores** do método `static` que retorna um array das constantes de `enum`.
- Quando uma constante `enum` for convertida em uma `String`, o identificador da constante é utilizado como a representação `String`.

```
1  // Figura 8.10: Book.java
2  // Declarando um tipo enum com um construtor e campos de
3  // instância explícitos 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", "2010" ),
9     CHTP( "C How to Program", "2007" ),
10    IW3HTP( "Internet & World Wide Web How to Program", "2008" ),
11    CPPHTP( "C++ How to Program", "2008" ),
12    VBHTP( "Visual Basic 2008 How to Program", "2009" ),
13    CSHARPHTP( "Visual C# 2008 How to Program", "2009" );
14
15    // campos de instância
16    private final String title; // título de livro
17    private final String copyrightYear; // ano dos direitos autorais
18
```

Constantes **enum** inicializadas com chamadas ao construtor

Figura 8.10 | Declarando um tipo **enum** com construtor e campos de instância explícitos e os métodos de acesso desses campos. (Parte 1 de 2.)

```
19       // construtor enum
20       Book( String bookTitle, String year )
21       {
22          title = bookTitle;
23          copyrightYear = year;
24       } // fim do construtor enum Book
25
26       // método de acesso para título de campo
27       public String getTitle()
28       {
29          return title;
30       } // fim do método getTitle
31
32       // método de acesso para o campo copyrightYear
33       public String getCopyrightYear()
34       {
35          return copyrightYear;
36       } // fim do método getCopyrightYear
37    } // fim do enum Book
```

Figura 8.10 | Declarando um tipo `enum` com construtor e campos de instância explícitos e os métodos de acesso desses campos. (Parte 2 de 2.)

```
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:\n" );
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.println( "\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    } // fim de main
23 } // fim da classe EnumTest
```

`values` do método `enum` retorna uma coleção de constantes enum

O método `EnumSet` `range` retorna uma coleção de constantes `enum` no intervalo especificado de constantes

Figura 8.11 | Testando um tipo `enum`. (Parte 1 de 2)

```
All books:

JHTP      Java How to Program                           2010
CHTP      C How to Program                              2007
IW3HTP    Internet & World Wide Web How to Program      2008
CPPHTP    C++ How to Program                            2008
VBHTP     Visual Basic 2008 How to Program              2009
CSHARPHTP Visual C# 2008 How to Program                 2009

Display a range of enum constants:

JHTP      Java How to Program                           2010
CHTP      C How to Program                              2007
IW3HTP    Internet & World Wide Web How to Program      2008
CPPHTP    C++ How to Program                            2008
```

Figura 8.11 | Testando um tipo **enum**. (Parte 2 de 2)

- Use o método `static` **range** da classe **EnumSet** (declarada no pacote `java.util`) para acessar um intervalo das constantes de `enum`.

  O método `range` aceita dois parâmetros — a primeira e a última constante `enum` no intervalo.

  Retorna um `EnumSet` que contém todas as constantes entre essas duas constantes, inclusive.
- A instrução `for` aprimorada pode ser usada com um `EnumSet` assim como com um array.
- A classe `EnumSet` fornece vários outros métodos `static`.

  `java.sun.com/javase/6/docs/api/java/util/EnumSet.html`.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.6

Em uma declaração enum*, é um erro de sintaxe declarar constantes* enum *depois dos construtores campos e métodos do tipo* enum*.*

8.10 Coleta de lixo e o método `finalize`

- Toda classe em Java tem os métodos da classe `Object` (pacote `java.lang`), um dos quais é o método `finalize`.

  Raramente utilizado porque pode causar problemas de desempenho e há uma incerteza sobre se ele será chamado.
- Todo objeto utiliza recursos do sistema, como memória.

  Precisamos de uma maneira disciplinada de devolver recursos para o sistema quando eles não são mais necessários; caso contrário, podem ocorrer vazamentos de recurso.
- A JVM realiza a **coleta de lixo automática** para reivindicar a memória ocupada por objetos que não são mais utilizados.

  Quando não houver mais referências a um objeto, o objeto está apto a ser coletado. Em geral, isso ocorre quando a JVM executa seu **coletor de lixo.**

- 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.
- Outros tipos de vazamentos de recursos podem ocorrer.

  Um aplicativo pode abrir um arquivo no disco para modificar os seus conteúdos.

  Se ele não fechar o arquivo, o aplicativo deve terminar antes de qualquer outro aplicativo pode utilizá-lo.

- O **método `finalize`** é chamado pelo coletor de lixo para realizar **limpeza de terminação** sobre um objeto um pouco antes de o coletor de lixo reivindicar a memória do objeto.

  O método `finalize` não aceita parâmetros e tem o tipo de retorno `void`.

  Um problema com relação ao método `finalize` é que não há garantias de o coletor de lixo executar em uma data/hora especificada.

  O coletor de lixo nunca pode executar antes de um programa terminar.

  Portanto, não fica claro se, ou quando, o método `finalize` será chamado.

  Por essa razão, a maioria dos programadores deve evitar o método `finalize`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.7

Uma classe que usa recursos de sistema, como arquivos no disco, deve fornecer um método que os programadores podem chamar para liberar recursos quando não forem mais necessários em um programa. Muitas classes da Java API fornecem métodos `close` *ou* `dispose` *para esse propósito. Por exemplo, a classe* `Scanner` (`java.sun.com/javase/6/docs/api/java/util/Scanner.html`) *tem um método* `close`.

8.11 Membros da classe `static`

- Em certos casos, apenas uma cópia de uma variável particular deve ser compartilhada por todos os objetos de uma classe.

  Um **field static** — chamado de **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`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.8

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

- Variáveis estáticas têm escopo de 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 private static de uma classe podem ser acessados pelo código do cliente somente por métodos da classe.
- Membros `static` de uma classe 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 quando 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 existir objetos da classe, forneça um método `public static` e chame-o qualificando seu nome com o nome de classe e um ponto.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.9

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

- Um método `static` não pode acessar membros de classe não `static`, 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 referenciar um objeto específico da classe, e quando um método `static` for chamado, poderia não haver nenhum objeto de sua classe na memória.
- Se uma variável `static` não for inicializada, o compilador atribuirá um valor padrão a essa variável — nesse caso `0`, o valor padrão para o tipo `int`.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.7

Um erro de compilação ocorre se um método `static` *chamar um método de instância (não* `static`*) na mesma classe utilizando apenas o nome do método. De modo 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.*

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.8

Referenciar `this` *em um método* `static` *é considerado um erro de sintaxe.*


```
1  // Figura 8.12: Employee.java
2  // Variável estática utilizada para manter uma contagem do número
3  // de objetos Employee na memória.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // o número de Employees criados
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 first, String last )
14    {
15       firstName = first;
16       lastName = last;
17
18       ++count;  // incrementa contagem estática de empregados
19       System.out.printf( "Employee constructor: %s %s; count = %d\n",
20          firstName, lastName, count );
21    } // fim do construtor Employee
22
```

Variável `static` compartilhada por todos os `Employee`s

Variáveis `static` podem ser acessadas por *todos* os métodos da classe

Figura 8.12 | Variável `static` utilizada para manter uma contagem do número de objetos `Employee` na memória. (Parte 1 de 2.)

```
23       // obtém o primeiro nome
24       public String getFirstName()
25       {
26          return firstName;
27       } // fim do método getFirstName
28
29       // obtém o último nome
30       public String getLastName()
31       {
32          return lastName;
33       } // fim do método getLastName
34
35       // método estático para obter valor de contagem estática
36       public static int getCount()
37       {
38          return count;
39       } // fim do método getCount
40    } // fim da classe Employee
```

Método `static` pode ser chamado pelos clientes da classe para obter a contagem atual — quer ou não haja qualquer objeto `Employee` na memória

Figura 8.12 | Variável `static` utilizada para manter uma contagem do número de objetos `Employee` na memória. (Parte 2 de 2.)

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 8.1

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

- Objetos `String` em Java são **imutáveis** — eles não podem ser modificados depois de criados. Portanto, é seguro ter muitas referências a um objeto `String`.

  Esse 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`.
- Operações de concatenação de `String` na verdade resultam na criação de um novo objeto `String` contendo o valor concatenado — 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.println( "\nEmployees after instantiation: " );
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
```

Obtém a contagem antes de criar `Employees`

Obtém a contagem depois de criar `Employees`; deve chamar métodos `static` somente via o nome da classe

Obtém a contagem depois de criar `Employees`

Figura 8.13 | A demonstração do membro `static`. (Parte 1 de 2.)

```
23          // obtém nomes de Employees
24          System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n",
25             e1.getFirstName(), e1.getLastName(),
26             e2.getFirstName(), e2.getLastName() );
27
28          // nesse exemplo, há somente uma referência a cada Employee,
29          // portanto as duas instruções seguintes indicam que
30          // esses objetos são elegíveis para a coleta de lixo
31          e1 = null;
32          e2 = null;
33       } // fim de main
34    } // fim da classe EmployeeTest
```

É uma boa prática configurar variáveis como `null` quando você não precisa mais dos objetos que elas referenciam; permite que o coletor de lixo recupere-as se não houver outras referências a esses objetos

```
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 | Demonstração do membro `static`. (Parte 2 de 2.)

- Os objetos tornam-se "elegíveis para a coleta de lixo" quando não há mais referências a eles no programa.
- Por fim, o coletor de lixo talvez reivindique a memória para esse objeto (ou o sistema operacional reivindicará a memória quando o programa terminar).
- A JVM não garante quando, ou até se, o coletor de lixo executará.
- Quando o coletor de lixo realmente executar, é possível que nenhum objeto ou apenas um subconjunto de objetos elegíveis seja coletado.

8.12 Importação `static`

- Uma declaração de **importação static** permite que você importe membros `static` de uma classe ou interface para que você possa acessá-los via os nomes não qualificados dos membros na sua classe — o nome de classe e um ponto (`.`) não devem utilizar um membro `static` importado.
- Duas formas:

  Uma que importa um membro `static` específico (o que é conhecido como **importação static simples**).

  Uma que importa todos os membros `static` de uma classe (o que é conhecido como **importação static sob demanda**).

- A sintaxe a seguir importa um membro `static` específico:

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

- onde *nomeDoPacote* é o pacote da classe, *NomeDaClasse* é o nome da classe, e *nomeDoMembroStatic* é o nome do campo ou método `static`.
- A sintaxe a seguir importa todos os membros `static` de uma classe:

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

- onde *nomeDoPacote* é o pacote da classe, e *NomeDaClasse* é o nome da classe.

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

```
1  // Figura 8.14: StaticImportTest.java
2  // Importação estática 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( "log( E ) = %.1f\n", log( E ) );
12       System.out.printf( "cos( 0.0 ) = %.1f\n", cos( 0.0 ) );
13    } // fim de main
14 } // fim da classe StaticImportTest
```

Permite que métodos Math sejam usados por seus nomes simples neste arquivo

```
sqrt( 900.0 ) = 30.0
ceil( -9.8 ) = -9.0
log( E ) = 1.0
cos( 0.0 ) = 1.0
```

Figura 8.14 | Importação estática dos métodos da classe Math.

Java™ COMO PROGRAMAR 8ª edição

DEITEL®

Erro comum de programação 8.9

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.*

8.13 Variáveis de instância `final`

- O **princípio do menor privilégio** é fundamental para uma boa engenharia de software.

  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.

  Torna os seus programas mais robustos evitando que o código modifique acidentalmente (ou maliciosamente) valores de variáveis e chame métodos que não devem ser acessíveis.
- 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.

```
private final int INCREMENT;
```

Declara uma variável de instância `final` `INCREMENT` (constante) do tipo `int`.

- Variáveis `final` podem ser inicializadas quando são declaradas pelos construtores da classe para que cada objeto da classe tenha um valor diferente.
- Se a classe fornecer múltiplos construtores, cada um deles deve inicializar a variável `final`.
- Uma variável `final` não pode ser modificada por atribuição depois que ela for inicializada.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.10

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 sendo* `final` *para evitar modificação.*

```
1  // Figura 8.15: Increment.java
2  // Variável de instância final em uma classe.
3
4  public class Increment
5  {
6     private int total = 0; // total de todos os incrementos
7     private final int INCREMENT; // variável constante (não inicializada)
8
9     // construtor inicializa variável de instância final INCREMENT
10    public Increment( int incrementValue )
11    {
12       INCREMENT = incrementValue; // inicializa variável constante (uma vez)
13    } // fim do construtor Increment
14
15    // adiciona INCREMENT ao total
16    public void addIncrementToTotal()
17    {
18       total += INCREMENT;
19    } // fim do método addIncrementToTotal
20
```

Variável final deve ser inicializada

Construtor realiza a inicialização

Figura 8.15 | Variável de instância final em uma classe. (Parte 1 de 2.)


```
21       // retorna representação de String dos dados de um objeto Increment
22       public String toString()
23       {
24          return String.format( "total = %d", total );
25       } // fim do método toString
26    } // fim da classe Increment
```

Figura 8.15 | Variável de instância `final` em uma classe. (Parte 2 de 2.)


```
1  // Figura 8.16: IncrementTest.java
2  // Variável final inicializada com um argumento de construtor.
3
4  public class IncrementTest
5  {
6     public static void main( String[] args )
7     {
8        Increment value = new Increment( 5 );
9
10       System.out.printf( "Before incrementing: %s\n\n", value );
11
12       for ( int i = 1; i <= 3; i++ )
13       {
14          value.addIncrementToTotal();
15          System.out.printf( "After increment %d: %s\n", i, value );
16       } // for final
17    } // fim de main
18 } // fim da classe IncrementTest
```

Argumento passado ao construtor para inicializar a variável de instância `final`

```
Before incrementing: total = 0

After increment 1: total = 5
After increment 2: total = 10
After increment 3: total = 15
```

Figura 8.16 | Variável `final` inicializada com um argumento de construtor.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.10

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

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 8.3

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 remover os bugs em tempo de compilação, se possível, em vez de permitir que passem para o tempo de execução (quando a correção, segundo estudos, costuma ser muito mais cara).*

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.11

Um campo `final` *também deve ser declarado* `static` *se ele for inicializado em sua declaração com um valor que é o mesmo de todos os objetos da classe. Depois dessa inicialização, seu valor não pode nunca 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`.

- Se uma variável `final` não for inicializada, ocorrerá um erro de compilação.


```
Increment.java:13: variable INCREMENT might not have been initialized
   } // fim do construtor de Increment
   ^
1 error
```

Figura 8.17 | A variável `final` `INCREMENT` deve ser inicializada.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.11

Não inicializar uma variável de instância `final` *na sua declaração ou em cada construtor da classe produz um erro de compilação indicando que a variável talvez não tenha sido inicializada.*

8.14 Estudo de caso da classe `Time`: criando pacotes

- Cada classe na Java API pertence a um pacote que contém um grupo de classes relacionadas.
- Pacotes são definidos uma vez, mas podem ser importados em muitos programas.
- Os pacotes ajudam a gerenciar a complexidade de componentes de um aplicativo.
- Os pacotes facilitam a reutilização de software permitindo que programas importem classes de outros pacotes, em vez de copiar as classes para cada programa que as utiliza.
- Os pacotes fornecem uma convenção para nomes únicos de classes que ajuda a evitar conflitos entre nomes de classe.

- Passos para criar uma classe reutilizável:
- Se a classe não for `public`, ela pode ser utilizada somente por outras classes no mesmo pacote.
- Escolha um nome de pacote e adicione uma **declaração `package`** ao arquivo de código-fonte para a declaração de classe reutilizável.

  Em cada arquivo de código-fonte do Java pode haver apenas uma declaração `package`, e ela deve preceder todas outras declarações e instruções.
- Compile a classe de modo que ela seja colocada no diretório de pacotes apropriado.
- Importe a classe reutilizável para um programa e utilize a classe.

- Colocar uma declaração `package` no início de um arquivo-fonte Java indica que a classe declarada no arquivo é parte do pacote especificado.
- Somente declarações `package`, declarações `import` e comentários podem aparecer fora das chaves de uma declaração de classe.
- Um arquivo de código-fonte Java deve ter a seguinte ordem:
  uma declaração `package` (se houver alguma),
  declarações `import` (if any) e, então,
  declarações de classe.
- Somente uma das declarações de classe em um arquivo particular pode ser `public`.
- Outras classes no arquivo são colocadas no pacote e podem ser utilizadas somente pelas outras classes no pacote.
- Classes não `public` estão em um pacote para suportar as classes reutilizáveis no pacote.


```
1  // Figura 8.18: Time1.java
2  // Declaração de classe Time1 mantém a hora no formato de 24 horas.
3  package com.deitel.jhtp.ch08;
4
5  public class Time1
6  {
7     private int hour; // 0 - 23
8     private int minute; // 0 - 59
9     private int second; // 0 - 59
10
11    // configura um novo valor de hora usando a hora universal;
12    // assegura que os dados permaneçam consistentes configurando valores inválidos como zero
13    public void setTime( int h, int m, int s )
14    {
15       hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // valida horas
16       minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // valida minutos
17       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // valida segundos
18    } // fim do método setTime
19
20    // converte em String no formato de hora universal (HH:MM:SS)
21    public String toUniversalString()
22    {
23       return String.format( "%02d:%02d:%02d", hour, minute, second );
24    } // fim do método do toUniversalString
```

Ajuda a tornar `Time1` um nome único de classe; deve ser a primeira instrução no arquivo

Figura 8.18 | Empacotando a classe `Time1` para reutilização. (Parte 1 de 2.)


```
25
26       // converte em String no formato padrão de hora (H:MM:SS AM ou PM)
27       public String toString()
28       {
29          return String.format( "%d:%02d:%02d %s",
30             ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
31             minute, second, ( hour < 12 ? "AM" : "PM" ) );
32       } // fim do método toString
33    } // fim da classe Time1
```

Figura 8.18 | Empacotando a classe `Time1` para reutilização. (Parte 2 de 2.)

- Cada nome de pacote deve iniciar com seu nome de domínio Internet na ordem inversa.

  Por exemplo, nosso nome de domínio é `deitel.com`, assim nossos nomes de pacotes iniciam com `com.deitel`.

  Para o nome de domínio *suafaculdade*`.edu`, o nome de pacote deve iniciar com `edu.`*suafaculdade*.
- Depois que o nome de domínio é invertido, você pode escolher qualquer outro nome para seu pacote.

  Escolhemos utilizar `jhtp` como o próximo nome em nosso nome de pacote para indicar que essa classe é do livro *Java, como programar.*

  O sobrenome em nosso nome de pacote especifica que esse pacote é para o Capítulo 8 (`ch08`).

- Compile a classe de modo que ela seja armazenada no diretório de pacotes apropriado.
- Quando um arquivo Java contendo uma declaração `package` é compilado, o arquivo de classe resultante é colocado no diretório especificado pela declaração.
- A declaração `package`

```
package com.deitel.jhtp.ch08;
```

indica que a classe `Time1` deve ser colocada no diretório

```
com
   deitel
      jhtp
         ch08
```

- Os nomes de diretório na declaração `package` especificam a posição exata das classes no pacote.

- A opção **-d** do comando `javac` faz com que o compilador `javac` crie diretórios apropriados com base declaração `package`.

  A opção também especifica onde os diretórios devem ser armazenados.
- Exemplo:

  `javac -d . Time1.java` para especificar que o primeiro diretório no nosso nome de pacote deve ser colocado no diretório atual(`.`).
- As classes compiladas são colocadas no diretório que é nomeado por último da instrução `package`.

- O nome `package` é parte do **nome de classe completamente qualificado.**
  O nome de classe `Time1` é na verdade `com.deitel.jhtp.ch08.Time1`
- Você pode utilizar esse nome completamente qualificado nos seus programas ou pode `import`ar a classe e usar seu **nome simples** (o nome de classe apenas).
- Se outro pacote também contiver uma classe `Time1`, os nomes de classe completamente qualificados podem ser utilizados para distinguir entre as classes no programa e impedir um **conflito de nomes** (também chamado de **colisão de nomes**).


```
1  // Figura 8.19: Time1PackageTest.java
2  // objeto Time1 utilizado em um aplicativo.
3  import com.deitel.jhtp.ch08.Time1; // importa a classe Time1
4
5  public class Time1PackageTest
6  {
7     public static void main( String[] args )
8     {
9        // cria e inicializa um objeto Time1
10       Time1 time = new Time1(); // chama o construtor Time1
11
12       // gera saída de representações de string da hora
13       System.out.print( "The initial universal time is: " );
14       System.out.println( time.toUniversalString() );
15       System.out.print( "The initial standard time is: " );
16       System.out.println( time.toString() );
17       System.out.println(); // gera saída de uma linha em branco
18
19       // altera a hora e gera saída da hora atualizada
20       time.setTime( 13, 27, 6 );
21       System.out.print( "Universal time after setTime is: " );
22       System.out.println( time.toUniversalString() );
23       System.out.print( "Standard time after setTime is: " );
24       System.out.println( time.toString() );
```

Importa a classe Time1 para uso neste arquivo de código-fonte

Figura 8.19 | Objeto `Time1` utilizado em um aplicativo. (Parte 1 de 2.)


```
25          System.out.println(); // gera saída de uma linha em branco
26
27          // configura hora com valores inválidos; gera saída da hora atualizada
28          time.setTime( 99, 99, 99 );
29          System.out.println( "After attempting invalid settings:" );
30          System.out.print( "Universal time: " );
31          System.out.println( time.toUniversalString() );
32          System.out.print( "Standard time: " );
33          System.out.println( time.toString() );
34       } // fim de main
35    } // fim da classe Time1PackageTest
```

```
The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

Figura 8.19 | Objeto `Time1` utilizado em um aplicativo. (Parte 2 de 2.)

- Na Figura 8.19, a linha 3 é uma **declaração de importação do tipo simples.**

  Ela especifica uma classe a importar.
- Se seu programa utilizar múltiplas classes no mesmo pacote, você poderá importar essas classes com uma **declaração de importação do tipo por demanda.**
- Exemplo:

```
import java.util.*; // importa classes java.util
```

  utiliza um asterisco (*) no fim da declaração `import` para informar o compilador de que todas as classes `public` no pacote `java.util` estão disponíveis para serem utilizadas no programa.

Somente as classes no pacote `java.util` utilizadas no programa são carregadas pela JVM.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.12

Utilizar a declaração `import import java.*;` *causa um erro de compilação. Você deve especificar o nome exato do pacote do qual você quer importar classes.*

- *Especificando o classpath durante a compilação*
- Ao compilar uma classe que usa classes de outros pacotes, `javac` deve localizar os arquivos `.class` para todas as outras classes que serão usadas.
- O compilador utiliza um objeto especial chamado **carregador de classe** para localizar as classes que ele precisa.

  O carregador de classe inicia pesquisando as classes Java padrão que são empacotadas no JDK.

  Ele então procura **pacotes opcionais.**

  Se a classe não for localizada nas classes Java padrão, nem nas classes de extensão, o carregador de classe pesquisa o **classpath**, que contém uma lista de locais em que classes são armazenadas.

- O classpath consiste em uma lista de diretórios ou **repositórios de arquivos**, cada um separado por um **separador de diretório.**

  Ponto-e-vírgula (`;`) no Windows ou dois-pontos (`:`) no UNIX/Linux/Mac OS X.
- Os repositórios de arquivos são arquivos individuais que contêm diretórios de outros arquivos, em geral, um formato compactado.

  Os repositórios de arquivos normalmente terminam com as extensões de nome de arquivo `.jar` ou `.zip`.
- Os diretórios e repositórios de arquivos especificados no classpath contêm as classes que você deseja disponibilizar para o compilador Java e JVM.

- Por padrão, o classpath consiste apenas no diretório atual.
- O classpath pode ser modificado
  1. fornecendo a opção **-classpath** para o compilador `javac`
  2. configurando a **variável de ambiente CLASSPATH** (não recomendado).
- Classpath

  `java.sun.com/javase/6/docs/technotes/tools/index.html#genera`

  A seção intitulada "Informações gerais" contém informações sobre a configuração do classpath para UNIX/Linux e Windows.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 8.13

Especificar um classpath explícito elimina o diretório atual do classpath. Isso impede que classes no diretório atual (incluindo pacotes no diretório atual) sejam carregadas adequadamente. Se classes precisarem ser carregadas do diretório atual, inclua um ponto (.) no classpath para especificar o diretório atual.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 8.12

Em geral, é mais adequado utilizar a opção `-classpath` *do compilador, em vez da variável de ambiente* `CLASSPATH`*, para especificar o classpath de um programa. Isso permite que cada aplicativo tenha seu próprio classpath.*

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 8.4

Especificar o classpath com a variável de ambiente CLASSPATH *pode resultar em erros sutis e difíceis de localizar em programas que usam diferentes versões do mesmo pacote.*

- *Especificando o classpath ao executar um aplicativo*
- Ao executar um aplicativo, a JVM deve ser capaz de localizar os arquivos `.class` utilizados nesse aplicativo.
- Como ocorre com o compilador, o comando `java` utiliza um carregador de classe que primeiro pesquisa as classes padrão e classes de extensão e, depois, pesquisa o classpath (o diretório atual por padrão).
- O classpath pode ser especificado explicitamente utilizando-se qualquer uma das técnicas discutidas para o compilador.
- Como ocorre com o compilador, é melhor especificar um classpath individual do programa via as opções de linha de comando da JVM.

  Se classes precisarem ser carregadas do diretório atual, inclua um ponto (`.`) ao classpath para especificar o diretório atual.

8.15 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**.
- 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 utilizado.


```
1  // Figura 8.20: PackageDataTest.java
2  // Membros de acesso de pacote de uma classe permanecem
3  // acessíveis a outras classes 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    } // fim de main
21 } // fim da classe PackageDataTest
22
```

Acessando variáveis de acesso de pacote na classe PackageData

Figura 8.20 | Os membros de acesso de pacote de uma classe permanecem acessíveis a outras classes no mesmo pacote. (Parte 1 de 3.)

```
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       } // fim do construtor PackageData
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       } // fim do método toString
41    } // fim da classe PackageData
```

A classe tem acesso de pacote; pode ser utilizado apenas por outras classes no mesmo diretório

Os dados de acesso de pacote somente podem ser acessados por outras classes no mesmo diretório que o pacote via uma referência a um objeto na classe

Figura 8.20 | Os membros de acesso de pacote de uma classe permanecem acessíveis a outras classes no mesmo pacote. (Parte 2 de 3.)

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

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

Figura 8.20 | Os membros de acesso de pacote de uma classe permanecem acessíveis a outras classes no mesmo pacote. (Parte 3 de 3.)

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

- O exemplo a seguir armazena informações sobre as formas exibidas de modo que possamos reproduzi-las toda vez que o sistema chamar `paintComponent`.
- Criamos classes "inteligentes" que podem desenhar a si próprias usando um objeto `Graphics`.
- A Figura 8.21 declara a classe `MyLine`, que tem todas essas capacidades.
- O método `paintComponent` na classe `DrawPanel` itera por um array de objetos `MyLine`.

  Cada iteração chama o método `draw` do objeto `MyLine` atual e passa a ele o objeto `Graphics` a desenhar no painel.


```
1  // Figura 8.21: MyLine.java
2  // Declaração da classe MyLine.
3  import java.awt.Color;
4  import java.awt.Graphics;
5
6  public class MyLine
7  {
8     private int x1; // coordenada x da 1a. extremidade
9     private int y1; // coordenada y da 1a. extremidade
10    private int x2; // coordenada x da 2a. extremidade
11    private int y2; // coordenada y da 2a. extremidade
12    private Color myColor; // cor dessa forma
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; // configura a coordenada x da 1a. extremidade
18       this.y1 = y1; // configura a coordenada y da 1a. extremidade
19       this.x2 = x2; // configura a coordenada x da 2a. extremidade
20       this.y2 = y2; // configura a coordenada y da 2a. extremidade
21       myColor = color; // configura a cor
22    } // fim do construtor MyLine
23
```

Figura 8.21 | A classe `MyLine` representa uma linha. (Parte 1 de 2.)


```
24       // Desenha a linha na cor especificada
25       public void draw( Graphics g )
26       {
27          g.setColor( myColor );
28          g.drawLine( x1, y1, x2, y2 );
29       } // fim do método draw
30    } // fim da classe MyLine
```

Uma `MyLine` sabe como desenhar-se via um objeto `Graphics` fornecido como um argumento

Figura 8.21 | A classe `MyLine` representa uma linha. (Parte 2 de 2.)


```
1  // Figura 8.22: 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.util.Random;
7  import javax.swing.JPanel;
8
9  public class DrawPanel extends JPanel
10 {
11    private Random randomNumbers = new Random();
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
```

Figura 8.22 | Criando objetos `MyLine` aleatórios. (Parte 1 de 3.)

```
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          } // for final
37       } // fim do construtor DrawPanel
38
```

Figura 8.22 | Criando objetos `MyLine` aleatórios. (Parte 2 de 3.)


```
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       } // fim do método paintComponent
48    } // fim da classe DrawPanel
```

Figura 8.22 | Criando objetos `MyLine` aleatórios. (Parte 3 de 3.)

```
1  // Figura 8.23: TestDraw.java
2  // Criando um JFrame para exibir um DrawPanel.
3  import javax.swing.JFrame;
4
5  public class TestDraw
6  {
7     public static void main( String[] args )
8     {
9        DrawPanel panel = new DrawPanel();
10       JFrame application = new JFrame();
11
12       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       application.add( panel );
14       application.setSize( 300, 300 );
15       application.setVisible( true );
16    } // fim de main
17 } // fim da classe TestDraw
```

Figura 8.23 | Criando um `JFrame` para exibir um `DrawPanel`. (Parte 2 de 2.)

Java™ COMO PROGRAMAR 8ª edição

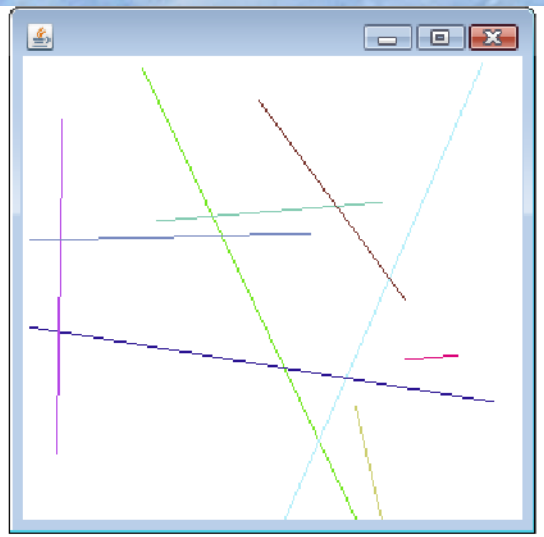


Figura 8.23 | Criando um `JFrame` para exibir um `DrawPanel`. (Parte 2 de 2.)