8

Classes e objetos: Um exame mais profundo

OBJETIVOS

- Neste capítulo, você aprenderá:
- Encapsulamento e ocultamento de dados.
- As noções de abstração de dados e tipos de dados abstratos (*abstract data types* – ADTs)
- Como utilizar a palavra-chave `this`.
- Como utilizar variáveis e métodos `static`.
- Como importar membros `static` de uma classe.
- Como utilizar o tipo `enum` para criar conjuntos de constantes com identificadores únicos.
- Como declarar constantes `enum` com parâmetros.

Sumário

Sumário

8.2 Estudo de caso da classe `Time`

- **Serviços `public` (ou interface `public`)**
  - **Métodos `public` disponíveis para uso por um cliente.**
- **Se uma classe não definir um construtor, o compilador fornecerá um construtor-padrão.**
- **Variáveis de instância:**
  - **Podem ser inicializadas quando são declaradas ou em um construtor.**
  - **Devem manter valores consistentes (válidos).**

Observação de engenharia de software 8.1

Os métodos que modificam os valores de variáveis `private` devem verificar se os novos valores pretendidos são adequados. Se não forem, os métodos *set* devem colocar as variáveis `private` em um estado consistente apropriado.

Resumo

Time1.java

(1 de 2)

```
1  // Fig. 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 data/hora usando UTC; assegura que
11    // os dados permaneçam consistentes configurando valores inválidos 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 ); // validate minutos
16       second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate segundos
17    } // fim do método setTime
18
```

Variáveis de instância **private**

Declara o método **public** **setTime**

Valida valores de parâmetro antes de configurar variáveis de instância

Resumo

Time1.java

(2 de 2)

```
19    // converte em String no formato de data/hora universal (HH:MM:SS)
20    public String toUniversalString()
21    {
22       return String.format( "%02d:%02d:%02d", hour, minute, second );
23    } // fim do método toUniversalString
24
25    // converte em String no formato padrão de data/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
```

Strings **format**

8.2 Estudo de caso da classe `Time` (*Continuação*)

- **Método `String format`:**
  - **Semelhante a `printf`, exceto que retorna uma string formatada em vez de exibi-la em uma janela de comando.**
- **`new` invoca implicitamente o construtor-padrão de `Time1`, uma vez que `Time1` não declara nenhum construtor.**

Observação de engenharia de software 8.2

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 direcionados 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.

Observação de engenharia de software 8.3

As interfaces mudam com menos freqüência 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.

Resumo

Time1Test.java

(1 de 2)

```
1  // Fig. 8.2: Time1Test.java
2  // Objeto Time1 usado 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 representações de string da data/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 branc
17
```

Cria um objeto **Time1**

Chama o método **toUniversalString**

Chama o método **toString**

Resumo

Time1Test.java

(2 de 2)

```
18       // altera a data/hora e gera saída da data/h
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
25
26       // configura data/hora com valores inválidos; gera saída da data/hora
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
```

Chama o método **setTime**

Chama o método **setTime** com valores inválidos

```
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
```

8.3 Escopo de classe

- **A interface pública de uma classe:**
  - **Métodos `public` apresentam uma visualização dos serviços que a classe fornece aos clientes da classe.**
- **Detalhes da implementação de uma classe:**
  - **Variáveis `private` e métodos `private` não são acessíveis aos clientes da classe.**

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.

Resumo

MemberAccessTest.java

```
1  // Fig. 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;     // error: hour has private access in Time1
10       time.minute = 15; // error: minute has private access in Time1
11       time.second = 30; // error: second has private access in Time1
12    } // fim de main
13 } // fim da classe MemberAccessTest
```

Tentando acessar variáveis de instância **private**

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

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

- **A referência `this`:**
  - **Qualquer objeto pode acessar uma referência dele mesmo com a palavra-chave `this`.**
  - **Métodos não-estáticos utilizam implicitamente `this` ao referenciar variáveis de instância do objeto e outros métodos.**
  - **Pode ser utilizada para acessar variáveis de instância quando elas estão sombreadas por variáveis locais ou parâmetros de método.**
- **Um arquivo `.java` pode conter mais de uma classe:**
  - **Mas somente uma classe em cada arquivo `.java` pode ser `public`.**

Resumo

ThisTest.java

(1 de 2)

```
1  // Fig. 8.4: ThisTest.java
2  // this utilizado implícita e explicitamente p/ 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    } // fi de main
11 } // fim da classe ThisTest
12
13 // class SimpleTime demonstrata 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 usa nomes de parâmetro idênticos aos
21    // nomes de variáveis de instância a referência "this" será
22    // necessária 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 o minuto do objeto "this"
27       this.second = second;   // configura o segundo do objeto "this"
28    } // fim do construtor SimpleTime
29
```

Cria um novo objeto **SimpleTime**

Declara variáveis de instância

Variáveis de instância sombreiam parâmetros de método

Utilizando `this` para acessar as variáveis de instância do objeto

Resumo

ThisTest.java

(2 de 2)

```
30    // utiliza "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
38    // converte em String no formato de data/hora universal (HH
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 toUniversalString
47 } // fim da classe SimpleTime
```

Utilizando **this** explícita e implicitamente para chamar **toUniversalString**

A utilização de **this** não é necessária aqui

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

Erro comum de programação 8.2

Freqüentemente 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.

Dica de prevenção de erro 8.1

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

Dica de desempenho 8.1

O Java economiza memória mantendo somente uma cópia de cada método por classe — esse método é invocado por todos os objetos 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:**
  - **Forneça múltiplas definições de construtor com diferentes assinaturas.**
- **Construtor sem argumento:**
  - **Um construtor invocado sem argumentos.**
- **A referência `this` pode ser utilizada para invocar um outro construtor:**
  - **Permitido somente como a primeira instrução no corpo de um construtor.**

Resumo

Time2.java

(1 de 4)

```
1  // Fig. 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: hora fornecida, minuto e segundo padronizados para 0
18    public Time2( int h )
19    {
20       this( h, 0, 0 ); // invoca o construtor Time2 com três argumentos
21    } // end Time2 one-argument constructor
22
23    // Construtor Time2: hora e minuto fornecidos, segundo padronizado para 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 um argumento Time2
28
```

Construtor sem argumento

Invoca construtor com três argumentos

Resumo

Time2.java

(2 de 4)

```
29    // Construtor Time2: hour, minute e second fornecidos
30    public Time2( int h, int m, int s )
31    {
32       setTime( h, m, s ); // invoca setTime para validar data/hora
33    } // fim do construtor Time2 de três argumentos
34
35    // Construtor Time2: outro objeto Time2 fornecido
36    public Time2( Time2 time )
37    {
38       // invoca construtor Time2 de tr
39       this( time.getHour(), time.getMinute(), time.getSecond() );
40    } // fim do construtor Time2 com um argumento de objeto Time2
41
42    // Métodos set
43    // configura um novo valor de data/hora usando UTC; assegura
44    // os dados permaneçam consistentes configurando 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
```

Chama o método **setTime**

Construtor recebe uma referência a um outro objeto **Time2** como um parâmetro

Poderia ter acessado diretamente variáveis de instância do objeto **time** aqui

Resumo

Time2.java

(3 de 4)

```
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    // validate and set minute
59    public void setMinute( int m )
60    {
61       minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
62    } // fim do método setMinute
63
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
```

Resumo

Time2.java

(4 de 4)

```
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
88
89    // converte em String no formato de data/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/hora (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
```

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 no construtor. Também é um erro de sintaxe se um método tentar invocar um construtor diretamente via `this`.

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. Utilizar variáveis de instância antes de elas serem inicializadas adequadamente é um erro de lógica.

Observação de engenharia de software 8.4

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`).

8.5 Estudo de caso da classe `Time`: Construtores sobrecarregados (*Cont.*)

- **Utilizando métodos *set*:**
  - **Fazer com que os construtores utilizem métodos *set* para modificar variáveis de instância, em vez de modificá-las diretamente, simplifica a alteração da implementação.**

Observação de engenharia de software 8.5

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.

Resumo

Chama construtores sobrecarregados

Time2Test.java

(1 de 3)

```
1  // Fig. 8.6: Time2Test.java
2  // Construtores 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
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
```

Resumo

Time2Test.java

(2 de 3)

```
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
```

```
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
```

Resumo

Time2Test.java

(3 de 3)

8.6 Construtores-padrão e sem argumentos

- **Cada classe deve ter pelo menos um construtor.**
  - **Se nenhum construtor for declarado, o compilador criará um construtor-padrão.**
    - **Não recebe nenhum argumento e inicializa variáveis de instância de acordo com seus valores iniciais especificados nas suas declarações ou de acordo com seus valores-padrão.**
      - **Valores-padrão são zero para tipos numéricos primitivos, `false` para valores `boolean` e `null` para referências.**
  - **Se os construtores forem declarados, a inicialização-padrão para objetos da classe será realizada por um construtor sem argumentos (se um for declarado).**

Erro comum de programação 8.5

Se uma classe tiver construtores, mas nenhum dos construtores `public` for um construtor sem argumentos, e um programa tentar chamar um construtor sem argumentos para inicializar um objeto da classe, ocorrerá um erro de compilação. Um construtor pode ser chamado sem argumentos somente se a classe não tiver nenhum construtor (nesse caso, o construtor-padrão é chamado) ou se a classe tiver um construtor `public` sem argumentos.

Observação de engenharia de software 8.6

O Java permite que outros métodos da classe além dos seus construtores tenham o mesmo nome da classe e que 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 os métodos que têm o mesmo nome da classe e que não especificam um tipo de retorno.

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

- **Métodos *set*:**
  - **Também conhecidos como métodos modificadores.**
  - **Atribuem valores a variáveis de instância.**
  - **Devem validar novos valores para variáveis de instância.**
    - **Podem retornar um valor para indicar dados inválidos.**
- **Métodos *get*:**
  - **Também conhecidos como métodos de acesso ou métodos de consulta.**
  - **Obtêm os valores das variáveis de instância.**
  - **Podem controlar o formato dos dados que retornam.**

Observação de engenharia de software 8.7

Se necessário, 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.

Observação de engenharia de software 8.8

Os projetistas de classe não precisam fornecer métodos *set* ou *get* para cada campo `private`. Essas capacidades devem ser fornecidas somente quando fizerem sentido.

8.7 Notas sobre os métodos *Set* e *Get* (*Continuação*)

- **Métodos predicados:**
  - **Testam se certa condição no objeto é verdadeira ou falsa e retornam o resultado.**
  - **Exemplo: Um método `isEmpty` para uma *classe contêiner* (uma classe capaz de conter muitos objetos).**
- **Encapsular tarefas específicas em métodos próprios simplifica os esforços de depuração.**

8.8 Composição

- **Composição:**
  - **Uma classe pode ter referências a objetos de outras classes como membros.**
  - **Às vezes conhecido como um relacionamento do tipo ‘*tem um*’.**

Observação de engenharia de software 8.9

Uma forma de reutilização de software é a composição, em que uma classe tem como membros referências a objetos de outras classes.

Resumo

Date.java

(1 de 3)

```
1  // Fig. 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
```

```
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    // 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
```

Valida o valor do mês

Valida o valor do dia

Resumo

Date.java

(2 de 3)

Resumo

Date.java

(3 de 3)

```
41       // verifica se day está no intervalo para month
42       if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
43          return testDay;
44
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 do 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
```

Verifique se o dia é 29 de fevereiro em um ano bissexto

Resumo

Employee.java

```
1  // Fig. 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 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
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
```

Employee contém referências a dois objetos **Date**

Chamadas implícitas aos métodos **hireDate** e **toString** de **birthDate**

Resumo

EmployeeTest.java

```
1  // Fig. 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
```

Cria um objeto **Employee**

Exibe o objeto **Employee**

```
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
```

8.9 Enumerações

- Tipos `enum`:
  - Declarados com uma declaração `enum`.
    - Uma lista separada por vírgulas de constantes `enum`.
    - Declara uma classe `enum` com as seguintes restrições:
      - tipos `enum` são implicitamente `final`;
      - constantes `enum` são implicitamente `static`; e
      - tentar criar um objeto de um tipo `enum` com `new` é um erro de compilação.
  - Constantes `enum` podem ser utilizadas em qualquer lugar em que as constantes podem ser utilizadas.
  - Construtor `enum`:
    - Como ocorre com construtores de classe, eles podem especificar parâmetros e podem ser sobrecarregados.

Resumo

Book.java

(1 de 2)

```
1  // Fig. 8.10: Book.java
2  // Declarando um tipo enum com um construtor e campos de instância explícitos
3  // e métodos de de acesso para esses campos
4
5  public enum Book
6  {
7     // declara constantes de tipo enum
8     JHTP6( "Java How to Program 6e", "2005" ),
9     CHTP4( "C How to Program 4e", "2004" ),
10    IW3HTP3( "Internet & World Wide Web How to Program 3e", "2004" ),
11    CPPHTP4( "C++ How to Program 4e", "2003" ),
12    VBHTP2( "Visual Basic .NET How to Program 2e", "2002" ),
13    CSHARPHTP( "C# How to Program", "2002" );
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 bookTitle, String year )
21    {
22       title = bookTitle;
23       copyrightYear = year;
24    } // fim do construtor enum Book
25
```

Declara seis constantes **enum**

Argumentos a serem passados para o construtor de **enum**

Declara variáveis de instância

Declara o construtor de **enum Book**

```
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
```

Resumo

Book.java

(2 de 2)

8.9 Enumerações (*Continuação*)

- **Método `static values`:**
  - **Gerado pelo compilador para cada `enum`.**
  - **Retorna um array das constantes de `enum` na ordem em que elas foram declaradas.**
- **Método `static range` da classe `EnumSet`:**
  - **Aceita dois parâmetros, a primeira e a última constantes `enum` no intervalo desejado.**
  - **Retorna um `EnumSet` que contém as constantes nesse intervalo, inclusive.**
  - **Uma instrução `for` aprimorada pode iterar por uma `EnumSet` assim como pode iterar por um array.**

Resumo

EnumTest.java

```
1  // Fig. 8.11: EnumTest.java
2  // Testando 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.JHTP6, Book.CPPHTP4 ) )
20          System.out.printf( "%-10s%-45s%s\n", book,
21             book.getTitle(), book.getCopyrightYear() );
22    } // fim de main
23 } // fim da classe EnumTest
```

Loop **for** aprimorado itera por cada constante **enum** no array retornado pelo método **value**

Loop **for** aprimorado itera por cada constante **enum** no **EnumSet** retornado pelo método **range**

```
All books:

JHTP6     Java How to Program 6e                          2005
CHTP4     C How to Program 4e                             2004
IW3HTP3   Internet & World Wide Web How to Program 3e     2004
CPPHTP4   C++ How to Program 4e                           2003
VBHTP2    Visual Basic .NET How to Program 2e             2002
CSHARPHTP C# How to Program                               2002

Display a range of enum constants:

JHTP6     Java How to Program 6e                          2005
CHTP4     C How to Program 4e                             2004
IW3HTP3   Internet & World Wide Web How to Program 3e     2004
CPPHTP4   C++ How to Program 4e                           2003
```

Resumo

EnumTest.java

(2 de 2)

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` na declaração `enum`.

8.10 Coleta de lixo e o método `finalize`

- **Coleta de lixo:**
  - **A JVM marca um objeto para coleta de lixo quando não há nenhuma outra referência a esse objeto.**
  - **O coletor de lixo da JVM irá recuperar esses objetos da memória para que eles possam ser utilizados por outros objetos.**
- **Método `finalize`:**
  - **Todas as classes em Java têm o método `finalize`:**
    - **Herdado da classe `Object`.**
  - **`finalize` é chamado pelo coletor de lixo quando ele realiza a faxina de terminação.**
  - **`finalize` não recebe nenhum parâmetro e tem tipo de retorno `void`.**

Observação de engenharia de software 8.10

Uma classe que utiliza recursos do sistema, como arquivos em disco, deve fornecer um método para finalmente liberar os recursos. Muitas classes da API do Java fornecem métodos `close` ou `dispose` para esse propósito.

Por exemplo, a classe `Scanner` (`java.sun.com/j2se/5.0/docs/api/java/util/Scanner.html`) tem um método `close`.

8.11 Membros da classe `static`

- Campos `static`:
  - Também conhecidos como variáveis de classe.
  - Representam informações de 'escopo de classe'.
  - Utilizados quando:
    - todos os objetos da classe devem compartilhar a mesma cópia dessa variável de instância; ou
    - essa variável de instância deve ser acessível mesmo quando não existir nenhum objeto da classe.
  - Podem ser acessados com o nome da classe ou com o nome de um objeto e um ponto (`.`).
  - Devem ser inicializados nas suas declarações ou, caso contrário, o compilador irá inicializá-los com um valor-padrão (`0` para `int`s).

Observação de engenharia de software 8.11

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

Observação de engenharia de software 8.12

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

Resumo

Employee.java

(1 de 2)

```
1  // Fig. 8.12: Employee.java
2  // Variável estática utilizada para manter uma contagem do número de
3  // Objetos Employee na memória.
4
5  public class Employee
6  {
7     private String firstName;
8     private String lastName;
9     private static int count = 0; // número de objetos na memória
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
```

Declara um campo **static**

Incrementa um campo **static**

Resumo

Employee.java

(2 de 2)

```
23    // subtrai 1 de static count quando o coletor de lixo
24    // chama finalize para limpar o objeto;
25    // confirma se finalize foi chamado
26    protected void finalize()
27    {
28       count--; // decrementa contagem estática de empregados
29       System.out.printf( "Employee finalizer: %s %s; count = %d\n",
30          firstName, lastName, count );
31    } // fim do método finalize
32
33    // obtém nome
34    public String getFirstName()
35    {
36       return firstName;
37    } // fim do método getFirstName
38
39    // obtém sobrenome
40    public String getLastName()
41    {
42       return lastName;
43    } // fim do método getLastName
44
45    // método static para obter  valor de contagem de estática
46    public static int getCount()
47    {
48       return count;
49    } // fim do método getCount
50 } // fim da classe Employee
```

Declara o método **finalize**

Declara o método **getCount** como **static** para obter o campo **static field**

Resumo

EmployeeTest.java

(1 de 3)

```
1  // Fig. 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 co
13       Employee e1 = new Employee( "Susan", "Baker" );
14       Employee e2 = new Employee( "Bob", "Blue" );
15
```

Chama o método **static getCount** utilizando o nome da classe **Employee**

Cria novos objetos **Employee**

Resumo

EmployeeTest.j

```
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
23      // obtém nomes de Employees
24      System.out.printf( "\nEmployee 1: %s %s\nEmployee 2: %s %s\n\n",
25         e1.getFirstName(), e1.getLastName(),
26         e2.getFirstName(), e2.getLastName() );
27
28      // nesse exemplo, há somente uma referência a cada Employee,
29      // ssim as duas instruções a seguir fazem com que a JVM marque cada
30      // objeto Employee para coleta de lixo
31      e1 = null;
32      e2 = null;
33
34      System.gc(); // pede que a coleta de lixo ocorra agora
35
```

Chama o método **static** **getCount** fora dos objetos

Chama o método **static** **getCount** dentro dos objetos

Remove referências aos objetos, a JVM irá marcá-las para coleta de lixo

Chama o método **static** **gc** da classe **System** para indicar que a coleta de lixo deve ser tentada

(2 de 3)

Resumo

EmployeeTest.java

(3 de 3)

```
36       // mostra a contagem de Employee depois de chamar o coletor de lixo; contagem
37       // ida pode ser 0, 1 ou 2 com base na execução do coletor de lixo
38       // imediata e número de objetos Employees coletados
39       System.out.printf( "\nEmployees after System.gc(): %d\n",
40          Employee.getCount() );
41    } // fim de main
42 } // fim da EmployeeTest
```

Chama o método **static** **getCount**

```
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

Employee finalizer: Bob Blue; count = 1
Employee finalizer: Susan Baker; count = 0

Employees after System.gc(): 0
```

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`.

8.11 Membros da classe `static` (*Continuação*)

- **Objetos `String` são imutáveis:**
  - **Operações de concatenação de string na verdade resultam na criação de um novo objeto `String`.**
- **Método `static gc` da classe `System`:**
  - **Indica que o coletor de lixo deve tentar da melhor maneira possível reivindicar objetos elegíveis para coleta de lixo.**
  - **É possível que nenhum objeto ou somente um subconjunto de objetos elegíveis seja coletado.**
- **Métodos `static` não podem acessar membros de classe não-`static`:**
  - **Também não podem utilizar a referência `this`.**

Erro comum de programação 8.7

Um erro de compilação ocorre se um método `static` chamar um método de uma instância (não-`static`) na mesma classe utilizando somente 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 somente o nome de variável.

Erro comum de programação 8.8

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

8.12 Import `static`

- **Declarações de importação `static`:**
  - **Permitem que os programadores façam referência a membros `static` importados como se eles fossem declarados na classe que os utiliza.**
  - **import `static` simples:**
    - **import `static nomeDoPacote.NomeDaClasse.nomeDoMembroStatic;`**
  - **import `static` por demanda:**
    - **import `static nomeDoPacote.NomeDaClasse.*;`**
    - **Importa todos os membros `static` da classe especificada.**

Resumo

StaticImportTest

.java

```
1  // Fig. 8.14: StaticImportTest.java
2  // Utilizando import static para importar métodos static 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
```

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

import **static** por demanda

Utiliza os métodos **static** e variáveis de instância de **Math** sem precedê-los com **Math**.

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`

- **Princípio do menor privilégio:**
  - **O código só deve ter o privilégio e acesso necessário para realizar sua tarefa e nada mais.**
- **Variáveis de instância `final`:**
  - **Palavra-chave `final`:**
    - **Especifica que uma variável não é modificável (é uma constante).**
  - **Variáveis de instância `final` podem ser inicializadas nas suas declarações.**
    - **Se não forem inicializadas nas suas declarações, elas deverão ser inicializadas em todos os construtores.**

Observação de engenharia de software 8.13

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.

Resumo

Increment.java

```
1  // Fig. 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
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 toIncrementString
26 } // fim da classe Increment
```

Declara a variável de instância **`final`**

Inicialize a variável de instância **`final`** dentro de um construtor

Resumo

```
1  // Fig. 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       } // fim de for
17    } // fim de main
18 } // fim da classe IncrementTest
```

IncrementTest.

Cria um objeto **Increment**

Chama o método **addIncrementToTotal**

```
Before incrementing: total = 0

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

Erro comum de programação 8.10

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

Dica de prevenção de erro 8.2

Tentativas de modificar uma variável de instância `final` são capturadas em tempo de compilação em vez de causar erros em tempo de execução.

Sempre é preferível remover bugs em tempo de compilação, se possível, em vez de permitir que eles passem para o tempo de execução (em que estudos descobriram que o custo do reparo é freqüentemente muito mais caro).

Observação de engenharia de software 8.14

Um campo `final` também deve ser declarado `static` se for inicializado na sua declaração. Depois que um campo `final` é inicializado na sua declaração, seu valor nunca pode mudar. Portanto, não é necessário criar 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`.

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. O mesmo erro ocorre se a classe inicializar a variável em alguns, mas não em todos, construtores da classe.

Resumo

```
Increment.java:13: variable INCREMENT might not have been initialized
   } // end Increment constructor
   ^
1 error
```

Increment.java

8.14 Capacidade de reutilização de software

- **Desenvolvimento rápido de aplicações:**
  - **A capacidade de reutilização de software acelera o desenvolvimento de softwares de alta qualidade e poderosos.**
- **A API do Java:**
  - **É uma estrutura em que os desenvolvedores Java podem trabalhar para conseguir verdadeira reutilização e rápido desenvolvimento de aplicações.**
  - **Documentação:**
    - **java.sun.com/j2se/5.0/docs/api/index.html**
      - **Ou java.sun.com/j2se/5.0/download.html para download.**

8.15 Abstração de dados e encapsulamento

- **Abstração de dados:**
  - **Ocultamento de informações.**
    - **Normalmente, classes ocultam os detalhes de implementação dos seus clientes.**
  - **Tipos de dados abstratos (Abstract Data Type – ADTs):**
    - **Representação de dados:**
      - **Exemplo: Tipo primitivo `int` é uma representação abstrata de um inteiro.**
        - **`int`s são apenas aproximações de inteiros; podem produzir estouros aritméticos.**
    - **Operações que podem ser realizadas nos dados.**

Boa prática de programação 8.2

Evite reinventar a roda. Estude as capacidades da API do Java. Se a API contiver uma classe que atenda aos requisitos do seu programa, utilize essa classe em vez de criar uma própria.

8.15 Abstração de dados e encapsulamento (*Continuação*)

- **Filas:**
  - **Semelhante a uma fila convencional:**
    - **Os clientes posicionam itens na fila (enfileiram um item).**
    - **Os clientes obtêm itens de volta a partir da fila (desenfileiram um item).**
    - **Ordem primeiro a entrar, primeiro a sair (first-in, first-out — FIFO).**
  - **A representação interna dos dados permanece oculta:**
    - **Os clientes só vêem a capacidade de enfileirar e desenfileirar itens.**

Observação de engenharia de software 8.15

Programadores criam tipos por meio do mecanismo de classe. Novos tipos podem ser projetados a fim de serem convenientes de utilizar como tipos predefinidos. Isso marca o Java como uma linguagem extensível. Embora a linguagem seja fácil de estender via os novos tipos, o programador não pode alterar a base da linguagem por conta própria.

8.16 Estudo de caso da classe `Time`: Controlando o acesso a membros

- **Para declarar uma classe como reutilizável:**
  - **Declare uma classe `public`.**
  - **Adicione uma declaração `package` ao arquivo de código-fonte:**
    - **deve ser a primeira instrução executável no arquivo;**
    - **o nome do `package` deve consistir no seu nome de domínio na Internet na ordem inversa seguido por outros nomes para o pacote.**
      - **Exemplo: `com.deitel.jhtp6.ch08`**
      - **O nome do `package` é parte do nome completamente qualificado de classe.**
        - **Distingue entre múltiplas classes que têm o mesmo nome, mas pertencem a pacotes diferentes.**
        - **Evita o conflito de nomes (também chamado *colisão de nomes*).**
      - **O nome da classe sem um nome do `package` é o nome simples.**

Resumo

Time1.java

```
1  // Fig. 8.18: Time1.java
2  // Declaração de classe Time1 mantém a data/hora no formato de 24 horas.
3  package com.deitel.jhtp6.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 data/hora usando data/hora universal; realiza
12    // testes de validade nos dados; configura 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
```

Declaração **package**

Time1 é uma classe **public**, portanto ela pode ser utilizada pelos importadores desse pacote

(1 de 2)

```
20    // converte em String no formato de data/hora universal (HH:MM:SS)
21    public String toUniversalString()
22    {
23       return String.format( "%02d:%02d:%02d", hour, minute, second );
24    } // fim do método toUniversalString
25
26    // converte em String no formato padrão de data/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
```

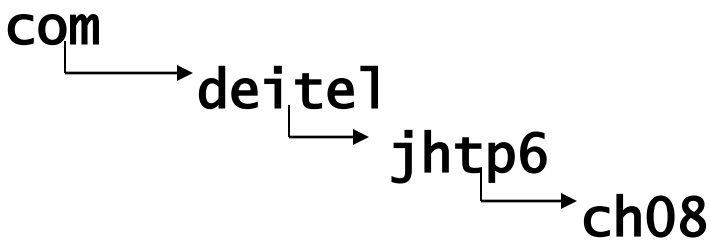
Resumo

Time1.java

(2 de 2)

8.16 Estudo de caso da classe `Time`: Controlando o acesso a membros (*Cont.*)

- **Compile a classe de modo que ela seja colocada na estrutura apropriada de diretórios de pacotes.**
  - **Exemplo: nosso pacote deveria estar no diretório:**



  - **Opção `-d` da linha de comando `javac`:**
    - **`javac` cria diretórios apropriados com base na declaração `package` da classe.**
    - **Um ponto (`.`) depois de `-d` representa o diretório atual.**

8.16 Estudo de caso da classe `Time`: Controlando o acesso a membros (*Cont.*)

- **Importe a classe reutilizável em um programa:**
  - **Declaração `import` do tipo simples.**
    - **Importa uma classe única.**
    - **Exemplo: `import java.util.Random;`**
  - **Declaração `import` do tipo por demanda:**
    - **Importa todas as classes em um pacote.**
    - **Exemplo: `import java.util.*;`**

Erro comum de programação 8.12

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

Resumo

```
1  // Fig. 8.19: Time1PackageTest.java
2  // Objeto Time1 utilizado em um aplicativo.
3  import com.deitel.jhtp6.ch08.Time1; // importa 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       // output string representations of the time
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
```

Declaração **`import`** do tipo simples

`Time1PackageTest.java`

Referência à classe **`Time1`** pelo seu nome simples

Resumo

Time1PackageTest

.java

(2 de 2)

```
19       // altera a data/hora e gera saída da data/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() );
25       System.out.println(); // gera saída de uma linha em branco
26
27       // configura data/hora com valores inválidos; gera saída da data/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
```

8.16 Estudo de caso da classe `Time`: Controlando o acesso a membros (*Cont.*)

- **Carregador de classe:**
  - **Localiza classes de que o compilador precisa:**
    - **Primeiro pesquisa as classes Java padrão empacotadas no JDK.**
    - **Depois procura pacotes opcionais.**
      - **Essas são permitidas pelo mecanismo de extensão do Java.**
    - **Por fim, pesquisa o `classpath`:**
      - **Lista de diretórios ou repositórios de arquivos separados por separadores de diretório.**
        - **Esses arquivos normalmente terminam com `.jar` ou `.zip`.**
        - **Classes-padrão estão no repositório de arquivos `rt.jar`.**

8.16 Estudo de caso da classe `Time`: Controlando o acesso a membros (*Cont.*)

- **Por padrão, o classpath consiste apenas no diretório atual. Mas ele pode ser modificado:**
  - **fornecendo a opção `-classpath` para o compilador `javac`; ou**
  - **configurando a variável de ambiente CLASSPATH.**
- **A JVM deve localizar as classes da mesma maneira como o compilador localiza:**
  - **O comando `java` pode utilizar outros `classpaths` com as mesmas técnicas utilizadas pelo comando `javac`.**

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 (.) ao classpath para especificar o diretório atual.

Observação de engenharia de software 8.16

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

Dica de prevenção de erro 8.3

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

8.17 Acesso de pacote

- **Acesso de pacote:**
  - **Métodos e variáveis declaradas sem nenhum modificador recebem acesso de pacote.**
  - **Isso não tem nenhum efeito se o programa consistir em uma única classe.**
  - **Isso tem efeito se o programa contiver múltiplas classes no mesmo pacote.**
    - **Membros com acesso de pacote podem ser acessados diretamente por meio de referências apropriadas aos objetos nas outras classes que pertencem ao mesmo pacote.**

Resumo

PackageDataTest.java

```
1  // Fig. 8.20: PackageDataTest.java
2  // Membros de acesso de pacote de uma classe permanecem acessíveis a outras
3  // 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
```

Pode acessar diretamente membros com acesso de pacote

Resumo

```
// classe com variáveis de instância de acesso de pacote
class PackageData
{
   int number; // variável de instância de acesso de pacote
   String string; // variável de instância de acesso de pacote

   // construtor
   public PackageData()
   {
      number = 0;
      string = "Hello";
   } // fim do construtor PackageData

   // retorna a representação String do objeto PackageData
   public String toString()
   {
      return String.format( "number: %d; string: %s", number, string );
   } // fim do método toString
} // fim da classe PackageData
```


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

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

PackageDataTest

.java

(2 de 2)

8.18 (Opcional) Estudo de caso de GUI e imagens gráficas: Utilizando objetos com imagens gráficas

- **Para criar um desenho consistente que permaneça idêntico todas as vezes que ele é desenhado:**
  - **Armazene informações sobre as formas exibidas de modo que elas possam ser reproduzidas exatamente da mesma maneira todas as vezes que o sistema chamar `paintComponent`.**

Resumo

Myline.java

```
1  // Fig. 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 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 myColor; // cor dessa forma
13
14    // construtor com valores de saída
15    public MyLine( int x1, int y1, int x2, int y2, Color color )
16    {
17       this.x1 = x1; // configura a coordenada x da primeira extremidade final
18       this.y1 = y1; // configura a coordenada y da primeira extremidade final
19       this.x2 = x2; // configura a coordenada x da segunda extremidade final
20       this.y2 = y2; // configura a coordenada y da segunda extremidade final
21       myColor = color; // configura a cor
22    } // fim do construtor MyLine
23
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
```

Variáveis de instância para armazenar coordenadas e cor de uma linha

Inicializa variáveis de instância

Desenha uma linha na cor adequada nas coordenadas adequadas

Resumo

DrawPanel.java

(1 de 2)

```
1  // Fig. 8.22: DrawPanel.java
2  // Programa que utiliza a classe MyLine
3  // para desenhas 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
```

Declara um array **MyLine**

Cria o array **MyLine**

Resumo

DrawPanel.java

(2 de 2)

```
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       } // fim de for
37    } // fim do construtor DrawPanel
38
39    // para cada array de forma, desenhas as formas individ
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
```

Gera coordenadas para essa linha

Gera uma cor para essa linha

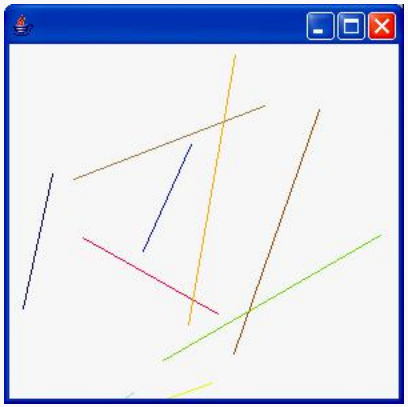
Cria o novo objeto **MyLine** com os atributos gerados

Desenha cada **MyLine**

Resumo

TestDraw.java

```
1  // Fig. 8.23: TestDraw.java
2  // Aplicativo de teste 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
```



8.19 (Opcional) Estudo de caso de engenharia de software: Começando a programar as classes do sistema ATM

- **Visibilidade:**
  - **Normalmente, atributos devem ser privados, e os métodos invocados pelos clientes devem ser públicos.**
  - ***Marcadores de visibilidade* em UML:**
    - **um sinal de adição (+) indica visibilidade pública; e**
    - **um sinal de subtração (-) indica visibilidade privada.**
- **Navegabilidade:**
  - **As *setas de navegabilidade* indicam em qual direção uma associação pode ser atravessada.**
  - **Navegabilidade *bidirecional*:**
    - **Associações com setas de navegabilidade nas duas extremidades ou nenhuma seta de navegabilidade pode ser atravessada em qualquer direção.**

8.19 Começando a programar as classes do sistema ATM (*Cont.*)

- **Implementando o sistema ATM a partir do seu design em UML (para cada classe):**
  - **Declare uma classe `public` com o nome no primeiro compartimento e um construtor sem argumentos vazio.**
  - **Declare variáveis de instância com base nos atributos no segundo compartimento.**
  - **Declare referências a outros objetos com base nas associações descritas no diagrama de classes.**
  - **Declare os shells dos métodos com base nas operações no terceiro compartimento.**
    - **Utilize o tipo de retorno `void` se nenhum tipo de retorno tiver sido especificado.**

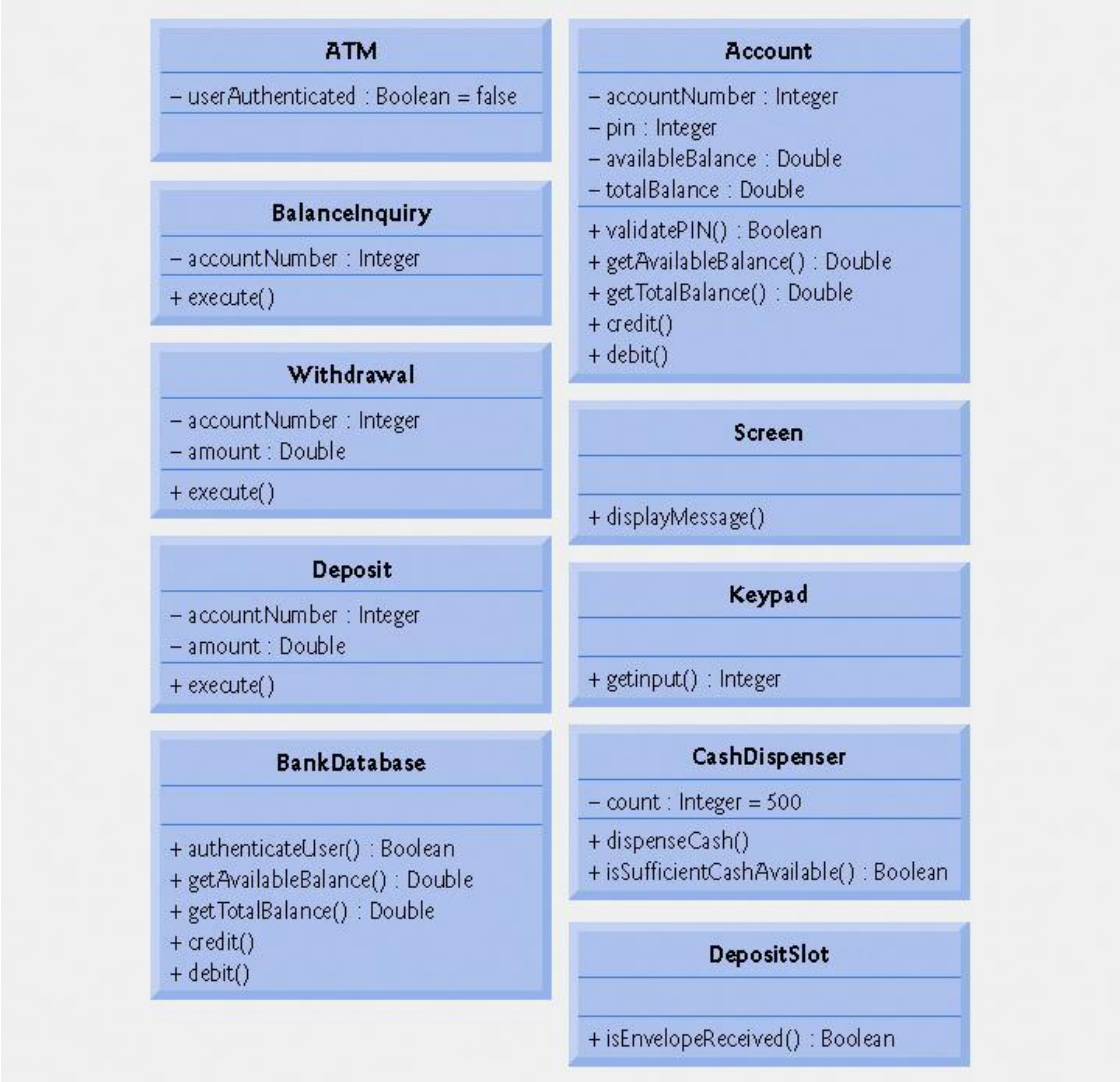


Figura 8.24 | Diagrama de classes com marcadores de visibilidade.

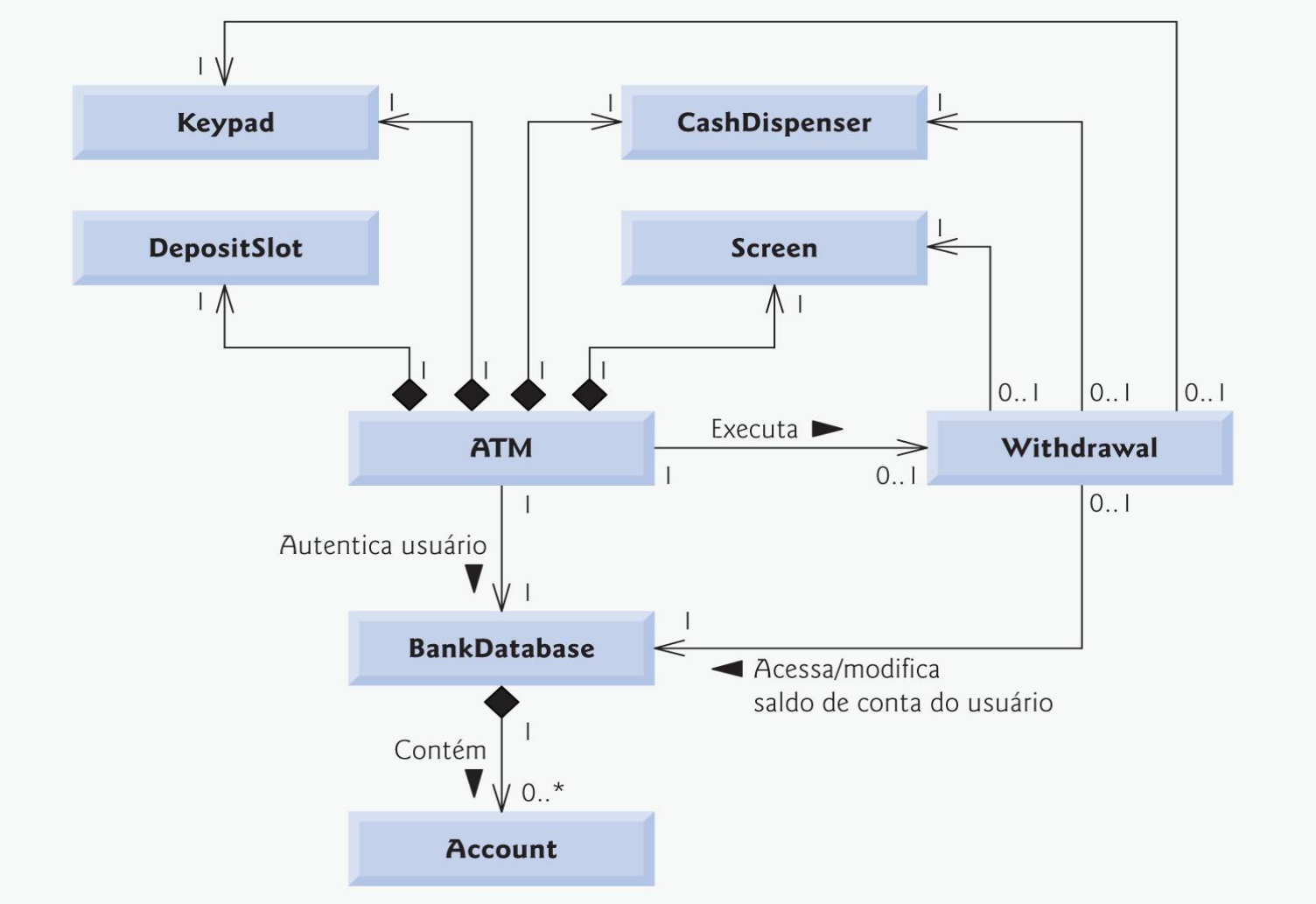


Figura 8.25 | Diagrama de classe com setas de navegabilidade.

```
1 // Classe Withdrawal representa uma transação de saque no ATM
2 public class Withdrawal
3 {
4    // construtor sem argumento
5    public Withdrawal()
6    {
7    } // fim do construtor sem argumentos Withdrawal
8 } // fim da classe Withdrawal
```

Classe para **Withdrawal**

Construtor sem argumento vazio

Outline

withdrawal.java

Resumo

withdrawal.java

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // atributos
5     private int accountNumber; // conta da qual a sacar fundos
6     private double amount; // quantia a retirar
7
8     // construtor sem argumento
9     public Withdrawal()
10    {
11    } // fim do construtor sem argumentos Withdrawal
12 } // fim da classe Withdrawal
```

Declara variáveis de instância

Resumo

withdrawal.java

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // atributos
5     private int accountNumber; // conta a sacar fundos
6     private double amount; // quantia a sacar
7
8     // referências a objetos associados
9     private Screen screen; // Tela do ATM
10    private Keypad keypad; // Teclado do ATM
11    private CashDispenser cashDispenser; // dispensador de cédulas do ATM
12    private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
13
14    // construtor sem argumentos
15    public Withdrawal()
16    {
17    } // fim do construtor Withdrawal sem argumentos
18 } // fim da classe Withdrawal
```

Declara referências a outros objetos

Resumo

withdrawal.java

```
1  // A classe Withdrawal representa uma transação de saque no ATM
2  public class Withdrawal
3  {
4     // atributos
5     private int accountNumber; // conta da qual sacar fundos
6     private double amount; // quantia a sacar
7
8     // referências a objetos associados
9     private Screen screen; // Tela do ATM
10    private Keypad keypad; // Teclado do ATM
11    private CashDispenser cashDispenser; // Dispensador de cédulas do ATM
12    private BankDatabase bankDatabase; // banco de dados de informações sobre a conta
13
14    // construtor sem argumentos
15    public Withdrawal()
16    {
17    } // fim do construtor sem argumentos Withdrawal
18
19    // operações
20    public void execute()
21    {
22    } // fim do método execute
23 } // fim da classe Withdrawal
```

Declara um shell de um método com um tipo de retorno **`void`**

Resumo

```
1  // A classe Keypad representa o teclado de um ATM
2  public class Keypad
3  {
4     // nenhum atributo foi especificado ainda
5
6     // construtor sem argumentos
7     public Keypad()
8     {
9     } // fim do construtor Keypad sem argumentos
10
11    // operações
12    public int getInput()
13    {
14    } // fim do método getInput
15 } // fim da classe Keypad
```

withdrawal.java