6

Métodos: Um exame mais profundo

OBJETIVOS

- Neste capítulo, você aprenderá:
- Como métodos e campos `static` são associados a uma classe inteira em vez de instâncias específicas da classe.
- Como utilizar métodos `Math` comuns disponíveis no API do Java.
- Os mecanismos para passar informações entre métodos.
- Como o mecanismo de retorno/chamada de método é suportado pela pilha de chamadas de métodos e registros de ativação.
- Como pacotes agrupam classes relacionadas.
- Como utilizar a geração de números aleatórios para implementar aplicativos de jogos de azar.
- Como a visibilidade das declarações é limitada a regiões específicas dos programas.
- O que é a sobrecarga de método e como criar métodos sobrecarregados.

Sumário

Sumário

6.1 Introdução

- **Técnica do ‘dividir para conquistar’:**
  - **Construa um programa grande a partir de partes menores (ou módulos).**
  - **Pode ser realizado utilizando métodos.**
- **Métodos `static` podem ser chamados sem a necessidade de um objeto da classe.**
- **Geração de número aleatório.**
- **Constantes.**

6.2 Módulos de programa em Java

- **Java Application Programming Interface (API):**
  - **Também conhecida como *API do Java* ou *biblioteca de classes Java*.**
  - **Contém os métodos e classes predefinidas:**
    - **Classes relacionadas são organizadas em pacotes.**
    - **Inclui os métodos para matemática, manipulações de strings/caracteres, entrada/saída, bancos de dados, rede, processamento de arquivos, verificação de erros e outros.**

Boa prática de programação 6.1

Familiarize-se com a rica coleção de classes e métodos fornecidos pela API do Java (`java.sun.com/j2se/5.0/docs/api/index.html`).

Na Seção 6.8, apresentamos uma visão geral dos vários pacotes comuns. No Apêndice G, explicamos como navegar pela documentação da API do Java.

Observação de engenharia de software 6.1

Não tente reinventar a roda. Quando possível, reutilize as classes e métodos na API do Java.

Isso reduz o tempo de desenvolvimento de programas e evita a introdução de erros.

6.2 Módulos de programa em Java (*Continuação*)

- **Métodos:**
  - **São denominados *funções* ou *procedimentos* nas outras linguagens.**
  - **Permitem ao programador modularizar os programas separando suas tarefas em unidades autocontidas.**
  - **Têm uma abordagem de ‘dividir para conquistar’.**
  - **São reutilizáveis em futuros programas.**
  - **Evitam repetição de código.**

Observação de engenharia de software 6.2

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

Esses métodos tornam mais fácil escrever, depurar, manter e modificar programas.

Dica de prevenção de erro 6.1

Um pequeno método que realiza uma tarefa é mais fácil de testar e depurar do que um método maior que realiza muitas tarefas.

Observação de engenharia de software 6.3

Se você não puder escolher um nome conciso que expresse a tarefa de um método, seu método talvez tente realizar um número excessivo de tarefas.

Em geral, é melhor dividir esse método em várias declarações de método menores.

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

- **Método `static` (ou método de classe)**
  - **Aplica-se à classe como um todo, em vez de a um objeto específico da classe.**
  - **Chame um método `static` utilizando a chamada de método:**
    NomeDaClasse`.`*nomeDoMétodo*`(` *argumentos* `)`
  - **Todos os métodos da classe `Math` são `static`:**
  - **Exemplo:** `.Math.sqrt(p900.0)`

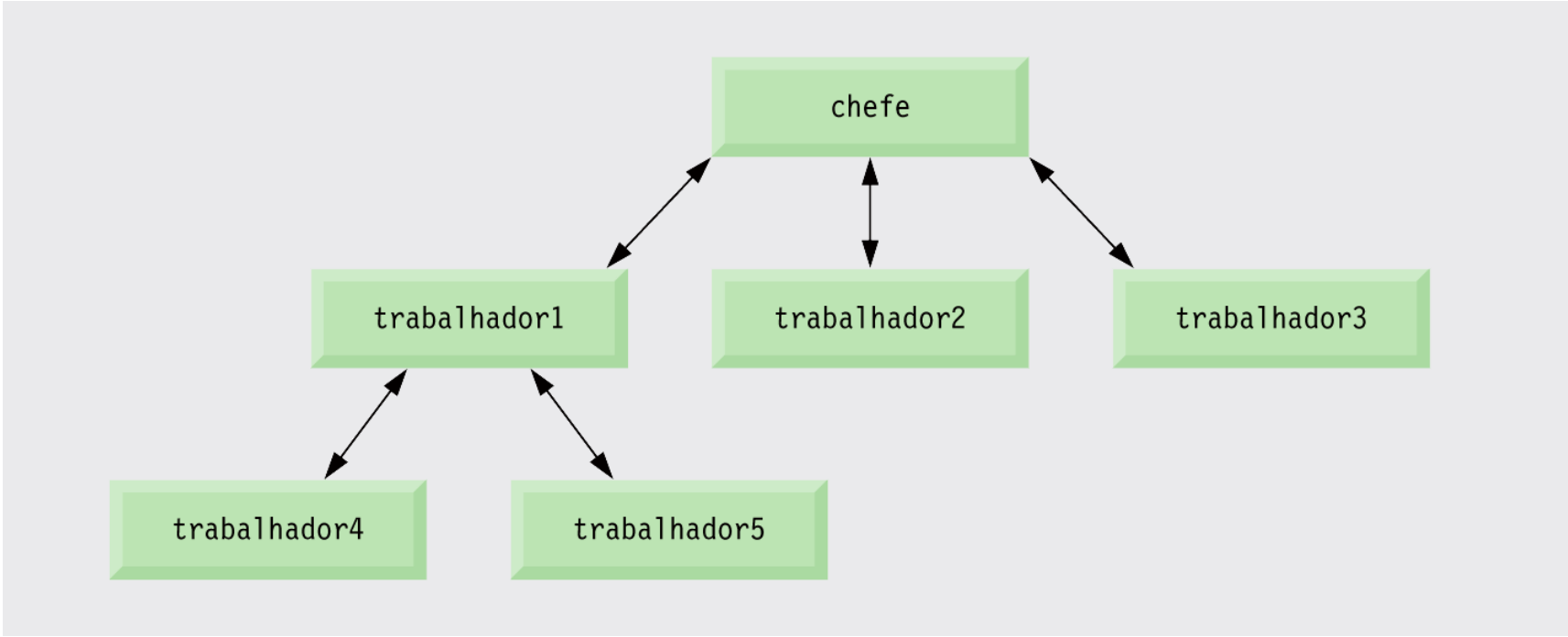


Figura 6.1 | Relacionamento hierárquico entre o método do trabalhador e o método do patrão.

Observação de engenharia de software 6.4

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

6.3 Métodos `static`, campos `static` e a classe `Math` (*Cont.*)

- **Constantes:**
  - **Palavra-chave `final`.**
  - **Não pode ser alterada depois da inicialização.**
- **Campos `static` (ou variáveis de classe):**
  - **São campos em que uma cópia da variável é compartilhada entre todos os objetos da classe.**
- **`Math.PI` e `Math.E` são campos `final static` da classe `Math`.**

Método	Descrição	Exemplo
`abs( x )`	valor absoluto de *x*	`abs( 23.7 )` é 23.7 `abs( 0.0 )` é 0.0 `abs( -23.7 )` é 23.7
`ceil( x )`	arredonda *x* para o menor inteiro não menor que *x*	`ceil( 9.2 )` é 10.0 `ceil( -9.8 )` é -9.0
`cos( x )`	co-seno trigonométrico de *x* (*x* em radianos)	`cos( 0.0 )` é 1.0
`exp( x )`	método exponencial e^x	`exp( 1.0 )` é 2.71828 `exp( 2.0 )` é 7.38906
`floor( x )`	arredonda *x* para o maior inteiro não maior que *x*	`Floor( 9.2 )` é 9.0 `floor( -9.8 )` é -10.0
`log( x )`	logaritmo natural de *x* (base *e*)	`log( Math.E )` é 1.0 `log( Math.E * Math.E )` é 2.0
`max( x, y )`	maior valor de *x* e *y*	`max( 2.3, 12.7 )` é 12.7 `max( -2.3, -12.7 )` é -2.3
`min( x, y )`	menor valor de *x* e *y*	`min( 2.3, 12.7 )` é 2.3 `min( -2.3, -12.7 )` é -12.7
`pow( x, y )`	*x* elevado à potência de *y* (isto é, x^y)	`pow( 2.0, 7.0 )` é 128.0 `pow( 9.0, 0.5 )` é 3.0
`sin( x )`	seno trigonométrico de *x* (*x* em radianos)	`sin( 0.0 )` é 0.0
`sqrt( x )`	raiz quadrada de *x*	`sqrt( 900.0 )` é 30.0
`tan( x )`	tangente trigonométrica de *x* (*x* em radianos)	`tan( 0.0 )` é 0.0

Figura 6.2 | Métodos da classe Math.

6.3 Métodos `static`, campos `static` e a classe `Math` (*Cont.*)

- **Método `main`:**
  - **`main` é declarado `static` para poder ser invocado sem criar um objeto da classe que contém `main`.**
  - **Qualquer classe pode conter um método `main`:**
    - **A JVM invoca o método `main` que pertence à classe especificada pelo primeiro argumento da linha de comando para o comando `java`.**

6.4 Declarando métodos com múltiplos parâmetros

- **Múltiplos parâmetros podem ser declarados especificando uma lista separada por vírgulas.**
  - **Os argumentos passados em uma chamada de método devem ser consistentes com o número, tipos e ordem dos parâmetros.**
    - **Às vezes, eles são chamados *parâmetros formais***

Resumo

MaximumFinder.java

(1 de 2)

```
1  // Fig. 6.3: MaximumFinder.java
2  // Método maximum declarado pelo programador.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtém três valores de ponto flutuante e localiza o valor máximo
8     public void determineMaximum()
9     {
10       // cria Scanner para entrada a partir da janela de comando
11       Scanner input = new Scanner( System.in );
12
13       // obtém a entrada do usuário
14       System.out.print(
15          "Enter three floating-point values separated by spaces: " );
16       double number1 = input.nextDouble(); // read first double
17       double number2 = input.nextDouble(); // read second double
18       double number3 = input.nextDouble(); // read third double
19
20       // determina o valor máximo
21       double result = maximum( number1, number2, number3 );
22
23       // exibe o valor máximo
24       System.out.println( "Maximum is: " + result );
25    } // fim do método determineMaximum
26
```

Solicita que o usuário insira e leia três valores **double**

Chama o método **maximum**

Exibe o valor máximo

```
27    // retorna o máximo dos seus três parâmetros de double
28    public double maximum( double x, double y, double z )
29    {
30       double maximumValue = x; // supõe que x é o maior valor inicial
31
32       // determina se y é maior que maximumValue
33       if ( y > maximumValue )
34          maximumValue = y;
35
36       // determina se z é maior que maximumValue
37       if ( z > maximumValue )
38          maximumValue = z;
39
40       return maximumValue;
41    } // fim do método maximum
42 } // fim da classe MaximumFinder
```

Declara o método **maximum**

MaximumFinder.java

Compara **y** e **maximumValue**

2)

Compara **z** e **maximumValue**

Retorna o valor máximo

Resumo

MaximumFinderTest.java

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

Cria um objeto **MaximumFinder**

Chama o método **determineMaximum**

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

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

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

Erro comum de programação 6.1

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

Observação de engenharia de software 6.5

Um método com muitos parâmetros pode estar realizando tarefas demais. Considere dividir o método em métodos menores que realizam tarefas separadas.

Como uma diretriz, se possível, tente ajustar o cabeçalho do método em uma linha.

6.4 Declarando métodos com múltiplos parâmetros (*Cont.*)

- **Reutilizando o método `Math.max`:**
  - **A expressão de `Math.max (x, Math.max (y, z))` determina o máximo de `y` e `z` e, então, determina o máximo de `x` e esse valor.**
- **Concatenação de string:**
  - **Utilizar o operador `+` com duas `String`s concatena-as em uma nova `String`.**
  - **Utilizar o operador `+` com uma `String` e um valor de um outro tipo de dados concatena `String` com uma representação de `String` do outro valor.**
    - **Se o outro valor for um objeto, seu método `toString` é chamado para gerar sua representação de `String`.**

Erro comum de programação 6.2

É um erro de sintaxe dividir uma literal de `String` em múltiplas linhas dentro de um programa.

Se uma string não couber em uma linha, divida a string em várias strings menores e utilize a concatenação para formar a `String` desejada.

Erro comum de programação 6.3

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

6.5 Notas sobre a declaração e utilização de métodos

- **Há três maneiras de chamar um método:**
  - **Utilize o nome de um método sozinho para chamar um outro método da mesma classe.**
  - **Utilize uma variável que contém uma referência a um objeto, seguido por um ponto (`.`) e o nome do método para chamar um método do objeto referenciado.**
  - **Utilize o nome da classe e um ponto (`.`) para chamar um método `static` de uma classe.**
- **Métodos `static` não podem chamar diretamente métodos não-`static` da mesma classe.**

6.5 Notas sobre a declaração e utilização de métodos (*Continuação*)

- **Há três maneiras de retornar o controle à instrução chamadora:**
  - **Se o método não retornar um resultado:**
    - **o fluxo do programa alcança a chave direita de fechamento do método; ou**
    - **o programa executa a instrução `return;`.**
  - **Se o método retornar um resultado:**
    - **O programa executa a *expressão* da instrução `return;`.**
      - **A *expressão* é primeiro avaliada e, então, seu valor é retornado ao chamador.**

Erro comum de programação 6.4

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

Erro comum de programação 6.5

Omitir o tipo-do-valor-de-retorno em uma declaração de método é um erro de sintaxe.

Erro comum de programação 6.6

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

Erro comum de programação 6.7

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

Erro comum de programação 6.8

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

6.6 Pilha de chamadas de método e registros de ativação

- **Pilhas:**
  - **Estruturas de dados do tipo último a entrar, primeiro a sair (*last-in, first-out* – LIFO):**
    - **Os itens são inseridos na parte superior.**
    - **Os itens são removidos da parte superior.**
- **Pilha de execução do programa:**
  - **Também conhecida como *pilha de chamadas de método*.**
  - **Endereços de retorno dos métodos chamadores são inseridos nessa pilha quando eles chamam outros métodos, e são removidos quando o controle retorna para eles.**

6.6 Pilha de chamadas de método e registros de ativação (*Continuação*)

- **As variáveis locais de um método são armazenadas em uma parte dessa pilha conhecida como *registro de ativação do método* ou *quadro (frame) de pilha*.**
  - **Quando a última variável que referencia um certo objeto é removida dessa pilha, esse objeto não mais permanece acessível pelo programa.**
    - **Ele por fim será excluído da memória durante a 'coleta de lixo'.**
  - **O estouro de pilha ocorre quando a pilha não pode alocar espaço suficiente para o registro de ativação de um método.**

6.7 Promoção e coerção de argumentos

- **Promoção de argumentos**
  - **O Java promoverá um argumento de chamada de método a fim de coincidir com seu parâmetro de método correspondente de acordo com as regras da promoção.**
  - **Os valores em uma expressão são promovidos para o tipo 'mais alto' na expressão (uma cópia temporária do valor é criada).**
  - **Converter valores para tipos mais baixos resulta em um erro de compilação, a menos que o programador faça com que a conversão ocorra explicitamente.**
    - **Coloque o tipo de dados desejado entre parênteses antes do valor (por exemplo: `(int)4.5)`.**

Tipo	Promoções válidas
double	Nenhuma
float	double
long	float ou double
int	long, float ou double
char	int, long, float ou double
shout	int, long, float ou double (mas não char)
byte	shout, int, long, float ou double (mas não char)
boolean	Nenhuma (os valores boolean não são considerados como números em Java)

Figura 6.5 | Promoções permitidas para tipos primitivos.

Erro comum de programação 6.9

Converter um valor de tipo primitivo em um outro tipo primitivo pode alterar o valor se o novo tipo não for uma promoção válida.

Por exemplo, converter um valor de ponto flutuante em um valor integral pode introduzir erros de truncamento (perda da parte fracionária) no resultado.

6.8 Pacotes da API do Java

- **Incluir a declaração**
  `import java.util.Scanner;`
  permite ao programador utilizar `Scanner` **em vez de** `java.util.Scanner`**.**
- **Documentação da API do Java:**
  - **java.sun.com/j2se/5.0/docs/api/index.html**
- **Visão geral dos pacotes no JDK 5.0:**
  - **java.sun.com/j2se/5.0/docs/api/overview-summary.html**

Pacote	Descrição
`java.applet`	O Java Applet Package contém uma classe e várias interfaces exigidas para criar applets Java — programas que executam nos navegadores da Web. (Os applets serão discutidos no Capítulo 20, Introdução a applets Java; e as interfaces no Capítulo 10, Programação orientada a objetos: Polimorfismo.)
`java.awt`	O Java Abstract Window Toolkit Package contém as classes e interfaces exigidas para criar e manipular GUIs no Java 1.0 e 1.1. Nas versões atuais do Java, os componentes GUI Swing dos pacotes javax.swing são freqüentemente utilizados em seu lugar. (Alguns elementos do pacote java.awt serão discutidos no Capítulo 11, Componentes GUI: Parte 1, Capítulo 12, Imagens gráficas e Java2D e Capítulo 22, Componentes GUI: Parte 2.)
`java.awt.event`	O Java Abstract Window Toolkit Event Package contém classes e interfaces que permitem o tratamento de eventos para componentes GUI tanto nos pacotes java.awt como `javax.swing`. (Você aprenderá mais sobre esse pacote no Capítulo 11: Componentes GUI: Parte 1 e no Capítulo 22: Componentes GUI: Parte 2.)
`java.io`	O Java Input/Output Package contém classes e interfaces que permitem aos programas gerar entrada e saída de dados. (Você aprenderá mais sobre esse pacote no Capítulo 14, Arquivos e fluxos.)
`java.lang`	O Java Language Package contém classes e interfaces (discutidas por todo esse texto) que são exigidas por muitos programas Java. Esse pacote é importado pelo compilador para todos os programas, assim o programador não precisa fazer isso.

Figura 6.6 | Pacotes da API do Java (um subconjunto). (Parte 1 de 2)

Pacote	Descrição
`java.net`	O Java Networking Package contém classes e interfaces que permitem aos programas comunicar-se via redes de computadores, como a Internet. (Você verá mais detalhes sobre isso no Capítulo 24, Redes.)
`java.text`	O Java Text Package contém classes e interfaces que permitem aos programas manipular números, datas, caracteres e strings. O pacote fornece recursos de internacionalização que permitem a um programa ser personalizado para um local específico (por exemplo, um programa pode exibir strings em diferentes idiomas com base no país do usuário).
`java.util`	O Java Utilities Package contém classes utilitárias e interfaces que permitem ações como manipulações de data e hora, processamento de números aleatórios (classe Random), armazenamento e processamento de grandes volumes de dados e a divisão de strings em parte menores chamadas tokens (classe `StringTokenizer`). (Você aprenderá mais sobre os recursos desse pacote no Capítulo 19, Coleções)
`javax.swing`	O Java Swing GUI Components Package contém classes e interfaces para componentes GUI Swing do Java que fornecem suporte para GUIs portáveis. (Você aprenderá mais sobre esse pacote no Capítulo 11: Componentes GUI: Parte 1 e no Capítulo 22: Componentes GUI: Parte 2.)
`javax.swing.event`	O Java Swing Event Package contém classes e interfaces que permitem o tratamento de eventos (por exemplo, responder a cliques de botão) para componentes GUI no pacote javax.swing. (Você aprenderá mais sobre esse pacote no Capítulo 11: Componentes GUI: Parte 1 e no Capítulo 22: Componentes GUI: Parte 2.)

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

Boa prática de programação 6.2

A documentação on-line da API do Java é fácil de pesquisar e fornece vários detalhes sobre cada classe. À medida que você aprende uma classe nesse livro, deve criar o hábito de examinar a classe na documentação on-line para informações adicionais.

6.9 Estudo de caso: Geração de números aleatórios

- **Geração de número aleatório:**
  - **Método `static` `random` da classe `Math`.**
    - **Retorna `doubles` no intervalo `0.0` <= x < `1.0`**
  - **Classe `Random` do pacote `java.util`:**
    - **Pode produzir números *pseudo-aleatórios* `boolean`, `byte`, `float`, `double`, `int`, `long` e gaussianos.**
    - **É semeado com a hora do dia atual a fim de gerar diferentes seqüências dos números toda vez que o programa é executado.**

Resumo

RandomIntegers.java
(1 de 2)

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

Importa a classe **Random** do pacote **java.util**

Cria um objeto **Random**

Gera uma rolagem aleatória de um dado

Resumo

`RandomIntegers.java`
(2 de 2)

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

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

Dois conjuntos diferentes de resultados contendo inteiros no intervalo 1-6

Resumo

RollDie.java

(1 de 3)

```
1  // Fig. 6.8: RollDie.java
2  // Rola um dado de seis lados 6000 vezes.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String args[] )
8     {
9        Random randomNumbers = new Random(); // gerador de número aleatório
10
11       int frequency1 = 0; // mantém a contagem de 1s lançado
12       int frequency2 = 0; // contagem de 2s lançados
13       int frequency3 = 0; // contagem de 3s lançados
14       int frequency4 = 0; // contagem de 4s lançados
15       int frequency5 = 0; // contagem de 5s lançados
16       int frequency6 = 0; // contagem de 6s lançados
17
```

Importa a classe **Random** do pacote **java.util**

Cria um objeto **Random**

Declara contadores de freqüência

Resumo

RollDie.java (2 de 3)

```
18       int face; // armazena o valor lançado mais recentemente
19
20       // resume os resultados de 6000 lançamentos de um da
21       for ( int roll = 1; roll <= 6000; roll++ )
22       {
23          face = 1 + randomNumbers.nextInt( 6 ); // número entre 1 a 6
24
25          // determina valor de lançamento de 1 a 6 e incrementa o contador apropriado
26          switch ( face )
27          {
28             case 1:
29                ++frequency1; // incrementa o contador de 1s
30                break;
31             case 2:
32                ++frequency2; // incrementa o contador de 2s
33                break;
34             case 3:
35                ++frequency3; // incrementa o contador de 3s
36                break;
37             case 4:
38                ++frequency4; // incrementa o contador de 4s
39                break;
40             case 5:
41                ++frequency5; // incrementa o contador de 5s
42                break;
43             case 6:
44                ++frequency6; // incrementa o contador de 6s
45                break; // opcional no final do switch
46          } // fim do switch
47       } // fim do for
48
```

Itera **6000** vezes

Gera uma rolagem aleatória de dados

switch baseado na rolagem dos dados

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

Exibe as freqüências da rolagem dos dados

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

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

Resumo

RollDie.java

(3 de 3)

6.9.1 Escalonamento e deslocamento generalizados de números aleatórios

- **Para gerar um número aleatório em certa seqüência ou intervalo:**
  - **Utilize a expressão**

    ***valorDeDeslocamento* + *diferençaEntreValores* * `randomNumbers.nextInt(` *fatorDeEscala* `)`**

    onde:
    - ***valorDeDeslocamento* é o primeiro número no intervalo desejado de valores;**
    - ***diferençaEntreValores* representa a diferença entre números consecutivos na seqüência; e**
    - ***fatorDeEscala* especifica quantos números estão no intervalo.**

6.9.2 Repetição de números aleatórios para teste e depuração

- **Para obter um objeto `Random` a fim de gerar a mesma seqüência de números aleatórios todas as vezes que o programa é executado, semeie-o com certo valor.**
  - **Ao criar o objeto `Random`:**
    ```
    Random randomNumbers =
      new Random( seedValue );
    ```
  - **Utilize o método `setSeed`:**
    ```
    randomNumbers.setSeed( seedValue );
    ```
  - **`seedValue` deve ser um argumento do tipo `long`.**

Dica de prevenção de erro 6.2

Enquanto um programa estiver sob desenvolvimento, crie o objeto `Random` com um valor específico de semente para produzir uma seqüência repetível de números aleatórios toda vez que o programa é executado. Se ocorrer um erro de lógica, corrija esse erro e teste o programa novamente com o mesmo valor de semente — isso permite reconstruir a mesma seqüência de números aleatórios que causou o erro. Depois que os erros de lógica forem removidos, crie o objeto `Random` sem utilizar um valor de semente, fazendo com que o objeto `Random` gere uma nova seqüência de números aleatórios toda vez que o programa é executado.

Resumo

Craps.java

(1 de 4)

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

Importa a classe **Random** do pacote **java.util**

Cria um objeto **Random**

Declara uma enumeração

Declara constantes

Resumo

Craps.java

(2 de 4)

```
20  // joga uma partida de craps
21  public void play()
22  {
23     int myPoint = 0; // pontos se não ganhar ou perder na 1a. rol
24     Status gameStatus; // pode conter CONTINUE, WON ou LOST
25
26     int sumOfDice = rollDice(); // primeira rolagem dos dados
27
28     // determina o status do jogo e a pontuação com base no primeiro lançamento
29     switch ( sumOfDice )
30     {
31        case SEVEN: // ganha com 7 no primeiro lançamento
32        case YO_LEVEN: // ganha com 11 no primeiro lançame
33           gameStatus = Status.WON;
34           break;
35        case SNAKE_EYES: // perde com 2 no primeiro lançamento
36        case TREY: // perde com 3 no primeiro lançamento
37        case BOX_CARS: // perde com 12 no primeiro lançamento
38           gameStatus = Status.LOST;
39           break;
40        default: // não ganhou nem perdeu, portanto registra a pontuaçao
41           gameStatus = Status.CONTINUE; // jogo não terminou
42           myPoint = sumOfDice; // armazena a pontuação
43           System.out.printf( "Point is %d\n", myPoint );
44           break; // opcional no final do switch
45     } // fim do switch
46
```

Chama o método **rollDice**

O jogador ganha com uma pontuação de 7 ou 11

O jogador perde com uma pontuação de 2, 3 ou 12

Configura e exibe a pontuação

Resumo

```
47      // enquanto o jogo não estiver completo
48      while ( gameStatus == Status.CONTINUE ) // nem WON nem LOST
49      {
50         sumOfDice = rollDice(); // lança os dados novamente
51
52         // determina o status do jogo
53         if ( sumOfDice == myPoint ) // vitória por pontuação
54            gameStatus = Status.WON;
55         else
56            if ( sumOfDice == SEVEN ) // perde obtendo 7 antes de atingir a pontuação
57               gameStatus = Status.LOST;
58      } // fim do while
59
60      // exibe uma mensagem de ganhou ou perdeu
61      if ( gameStatus == Status.WON )
62         System.out.println( "Player wins" );
63      else
64         System.out.println( "Player loses" );
65   } // fim do método play
66
```

Chama o método **rollDice**

Craps.java

O jogador ganha fazendo a pontuação

O jogador perde rolando 7

Exibe o resultado

Resumo

Craps.java

(4 de 4)

```
67    // lança os dados, calcula a soma e exibe os resultados
68    public int rollDice()
69    {
70       // seleciona valores aleatórios do dado
71       int die1 = 1 + randomNumbers.nextInt( 6 ); // rolagem do primeiro dado
72       int die2 = 1 + randomNumbers.nextInt( 6 ); // rolagem do segundo dado
73
74       int sum = die1 + die2; // soma dos valores dos dados
75
76       // exibe os resultados desse lançamento
77       System.out.printf( "Player rolled %d + %d = %d\n",
78          die1, die2, sum );
79
80       return sum; // retorna a soma dos dados
81    } // fim do método rollDice
82 } // fim da classe Craps
```

Declare o método `rollDice`

Gera rolagens de dois dados

Exibe as rolagens dos dados e suas somas

Resumo

CrapsTest.java

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

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

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

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

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

6.10 Estudo de caso: Um jogo de azar (introdução a enumerações)

- **Enumerações:**
  - **Tipos declarados pelo programador consistem em conjuntos de constantes.**
  - **Palavra-chave `enum`.**
  - **Um nome de tipo (por exemplo, `Status`).**
  - **Constantes de enumeração (por exemplo, `WON`, `LOST` e `CONTINUE`).**
    - **Não podem ser comparadas contra `int`s.**

Boa prática de programação 6.3

Utilize somente letras maiúsculas nos nomes das constantes. Isso faz com que as constantes sejam destacadas em um programa e lembre o programador de que constantes de enumeração não são variáveis.

Boa prática de programação 6.4

Utilizar constantes de enumeração (como `Status.WON`, `Status.LOST` e `Status.CONTINUE`) em vez de valores literais inteiros (como 0, 1 e 2) pode tornar os programas mais fáceis de ler e manter.

6.11 Escopo das declarações

- **Regras de escopo básicas:**
  - **O escopo de uma declaração de parâmetro é o corpo do método em que aparece.**
  - **O escopo da declaração de uma variável local ocorre do ponto da declaração até o final desse bloco.**
  - **O escopo da declaração de uma variável local na seção de inicialização de um cabeçalho `for` é o restante do cabeçalho `for` e o corpo da instrução `for`.**
  - **O escopo de um método ou campo de uma classe é o corpo inteiro da classe.**

6.11 Escopo das declarações (*Continuação*)

- **Sombreamento:**
  - **Um campo é sombreado (ou oculto) se uma variável local ou parâmetro tiver o mesmo nome do campo.**
    - **Isso permanece até a variável local ou parâmetro sair do escopo.**

Erro comum de programação 6.10

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

Dica de prevenção de erro 6.3

Utilize nomes diferentes para campos e variáveis locais para ajudar a evitar erros de lógica sutis que ocorrem quando um método é chamado e uma variável local do método sombreia um campo com o mesmo nome na classe.

Resumo

Scope.java

(1 de 2)

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

Sombreia o campo **x**

Exibe o valor da variável local **x**

Resumo

Scope.java

(2 de 2)

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

Sobreia o campo **x**

Exibe o valor da variável local **x**

Exibe o valor do campo **x**

Resumo

ScopeTest.java

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

```
local x in method begin is 5

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

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

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

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

local x in method begin is 5
```

6.12 Sobrecarga de método

- **Sobrecarga de método**
  - **Múltiplos métodos com o mesmo nome, mas diferentes tipos, número ou ordem dos parâmetros nas suas listas de parâmetros.**
  - **O compilador decide qual método está sendo chamado comparando uma lista dos argumentos da chamada de método com uma das listas de parâmetros dos métodos sobrecarregados.**
    - **A ordem, nome, número, tipo dos parâmetros de um método formam sua assinatura.**
  - **As diferenças no tipo de retorno são irrelevantes na sobrecarga de método.**
    - **Métodos sobrecarregados podem ter diferentes tipos de retorno.**
    - **Métodos com diferentes tipos de retorno, mas com a mesma assinatura, resultam em um erro de compilação.**

Resumo

```
1  // Fig. 6.13: MethodOverload.java
2  // Declarações de métodos sobrecarregados.
3
4  public class MethodOverload
5  {
6     // testa métodos square sobrecarregados
7     public void testOverloadedMethods()
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end method testOverloadedMethods
12
13    // método square com argumento int
14    public int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int
17          intValue );
18       return intValue * intValue;
19    } // fim do método square com argumento int
20
21    // método square com argumento double
22    public double square( double doubleValue )
23    {
24       System.out.printf( "\nCalled square with double argument: %f\n",
25          doubleValue );
26       return doubleValue * doubleValue;
27    } // fim do método square com argumento double
28 } // fim da classe MethodOverload
```

Chama corretamente o método "**square** de **int**"

Chama corretamente o método "**square** de **double**"

Declarando o método "**square** de **int**"

Declarando o método "**square** de **double**"

MethodOverload.java

Resumo

MethodOverloadTest.java

```
1  // Fig. 6.14: MethodOverloadTest.java
2  // Aplicativo para testar a classe MethodOverload.
3
4  public class MethodOverloadTest
5  {
6     public static void main( String args[] )
7     {
8        MethodOverload methodOverload = new MethodOverload();
9        methodOverload.testOverloadedMethods();
10    } // fim de main
11 } // fim da classe MethodOverloadTest
```

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

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

Resumo

MethodOverload Error.java

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

Mesma assinatura de método

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

Erro de compilação

Erro comum de programação 6.11

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

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

- Classe `Color` do pacote `java.awt`
  - Representada como valores RBG (red, green e blue — vermelho, verde e azul):
    - Cada componente tem um valor de 0 a 255.
  - 13 objetos `static Color` predefinidos:
    - `Color.Black`, `Coor.BLUE`, `Color.CYAN`, `Color.DARK_GRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LIGHT_GRAY`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE` e `Color.YELLOW`

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

- **Métodos `fillRect` e `fillOval` da classe `Graphics`:**
  - **Semelhante a `drawRect` e `drawOval`, mas desenha retângulos e ovais preenchidos com cores:**
    - **Os dois primeiros parâmetros especificam as coordenadas do canto superior esquerdo e os dois segundos parâmetros especificam a largura e altura.**
- **Método `setColor` da classe `Graphics`:**
  - **Configure a cor do desenho atual (para preencher retângulos e ovais desenhados por `fillRect` e `fillOval`).**

Resumo

DrawSmiley.java

```
1  // Fig. 6.16: DrawSmiley.java
2  // Demonstra formas preenchidas.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DrawSmiley extends JPanel
8  {
9     public void paintComponent( Graphics g )
10    {
11       super.paintComponent( g );
12
13       // desenha o rosto
14       g.setColor( Color.YELLOW );
15       g.fillOval( 10, 10, 200, 200 );
16
17       // desenha os olhos
18       g.setColor( Color.BLACK );
19       g.fillOval( 55, 65, 30, 30 );
20       g.fillOval( 135, 65, 30, 30 );
21
22       // desenha a boca
23       g.fillOval( 50, 110, 120, 60 );
24
25       // "retoca" a boca para criar um sorriso
26       g.setColor( Color.YELLOW );
27       g.fillRect( 50, 110, 120, 30 );
28       g.fillOval( 50, 120, 120, 40 );
29    } // fim do método paintComponent
30 } // fim da classe DrawSmiley
```

Classe `Color`

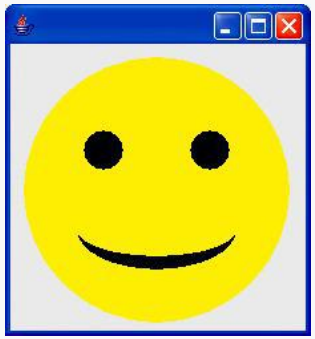
Configura as cores de preenchimento

Desenha formas preenchidas

Resumo

DrawSmileyTest.java

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



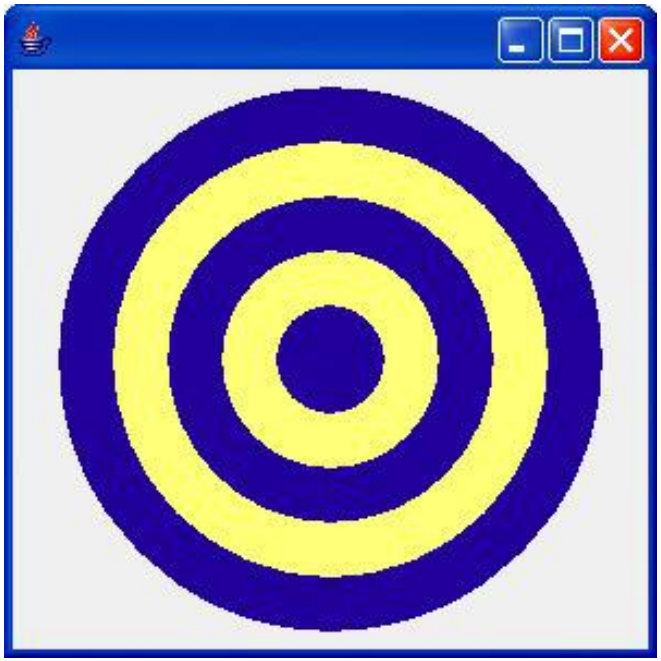


Figura 6.18 | Um alvo com duas cores aleatórias alternantes.

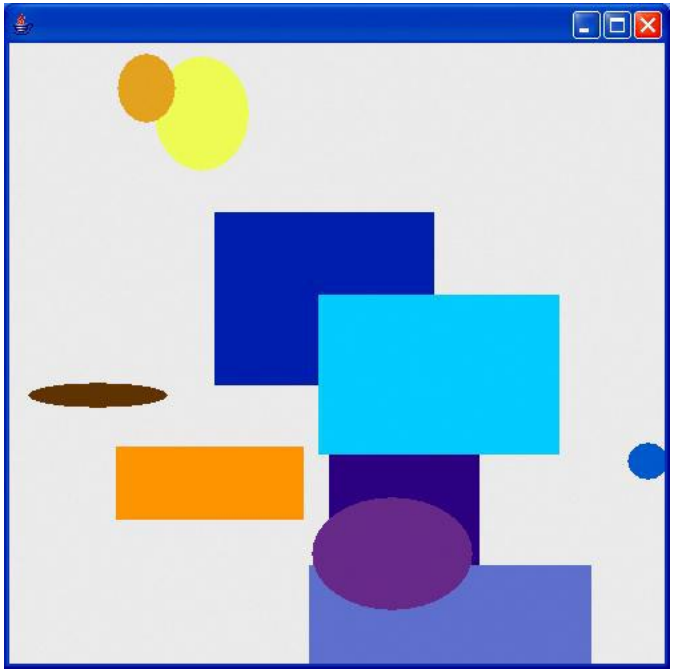


Figura 6.19 | Formas aleatoriamente geradas.

6.14 (Opcional) Identificando operações de classe

- **Identificando operações:**
  - **Examine os verbos-chave e frases com verbo no documento de requisitos.**
- **Modela as operações em UML:**
  - **Cada operação recebe um nome de operação, uma lista de parâmetros e um tipo de retorno:**
    - ***nomeDaOperação*（*parâmetro1*, *parâmetro2*, …, *parâmetroN*）: *tipo de retorno***
  - **Cada parâmetro tem um nome de parâmetro e um tipo de**
    - ***nomeDoParâmetro*: *tipoDoParâmetro***

6.14 (Opcional) Identificando operações de classe (*Continuação*)

- **Algumas operações talvez ainda não tenham tipos de retorno:**
  - **Os tipos de retorno restantes serão adicionados à medida que o projeto e a implementação vão avançando.**
- **Identificando e modelando parâmetros de operação:**
  - **Examine quais dados a operação requer para realizar sua tarefa atribuída.**
  - **Parâmetros adicionais podem ser adicionados mais tarde.**

Class	Verbos e frases com verbos
ATM	executar transações financeiras
BalanceInquiry	[nenhuma no documento de requisitos]
Withdrawal	[nenhuma no documento de requisitos]
Deposit	[nenhuma no documento de requisitos]
BankDatabase	autentica um usuário, recupera um saldo em conta, credita uma quantia depositada em uma conta, debita um valor sacado de uma conta
Account	recupera um saldo em conta, credita uma quantia depositada em uma conta, debita uma quantia sacada de uma conta
Screen	exibe uma mensagem para o usuário
Keypad	recebe entrada numérica do usuário
CashDispenser	fornece o dinheiro, indica se contém dinheiro suficiente para satisfazer uma solicitação de saque
DepositSlot	recebe um envelope de depósito

Figura 6.20 | Os verbos e frases com verbo para cada classe no sistema ATM.

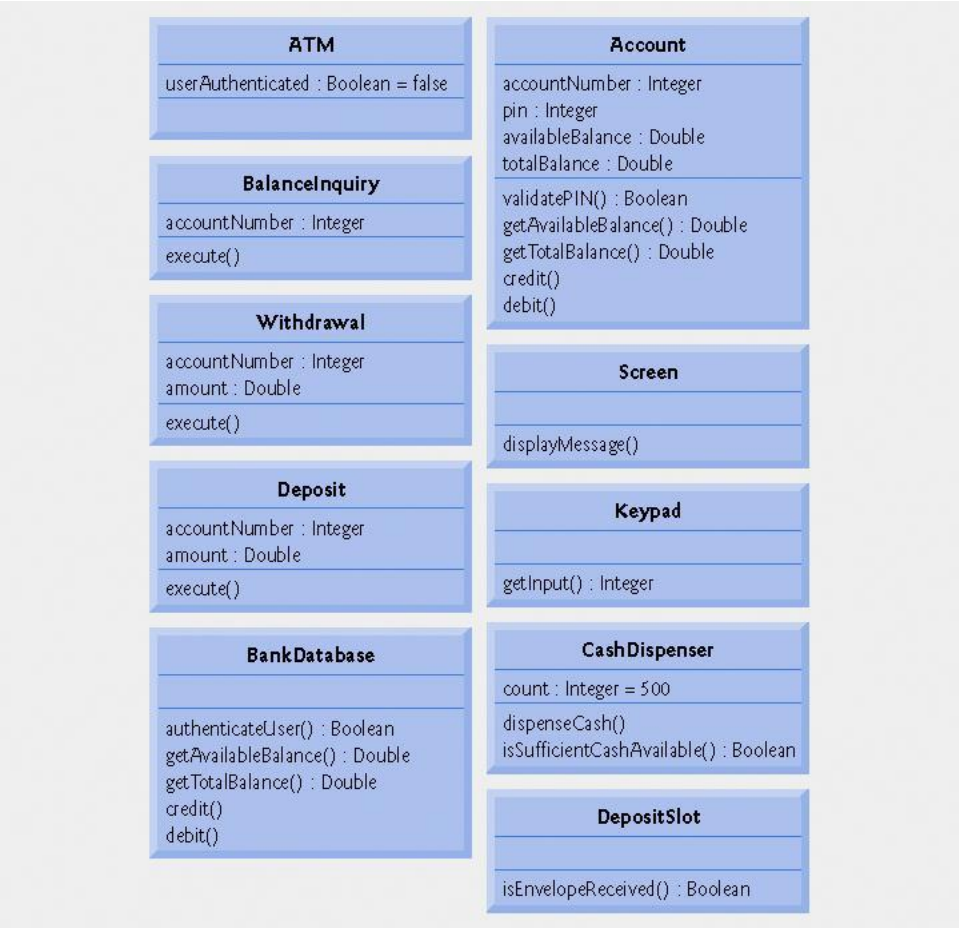


Figura 6.21 | Classes no sistema ATM com atributos e operações.

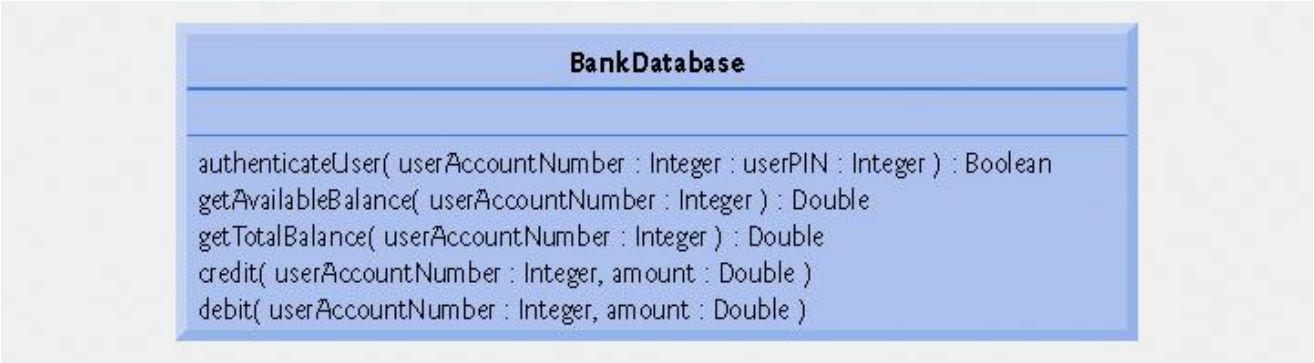


Figura 6.22 | Classe `BankDatabase` com parâmetros de operação.

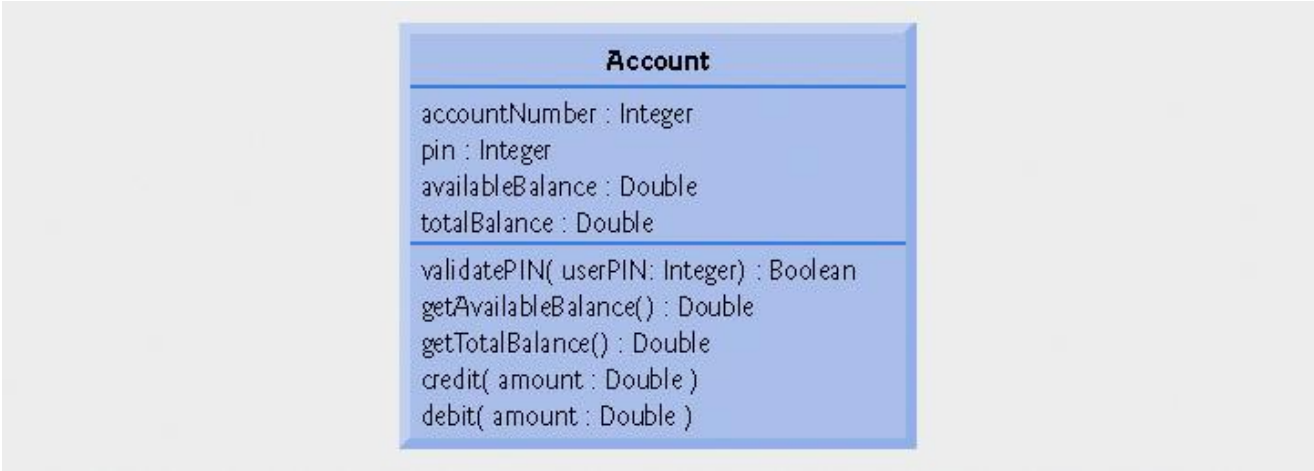


Figura 6.23 | Classe `Account` com parâmetros de operação.

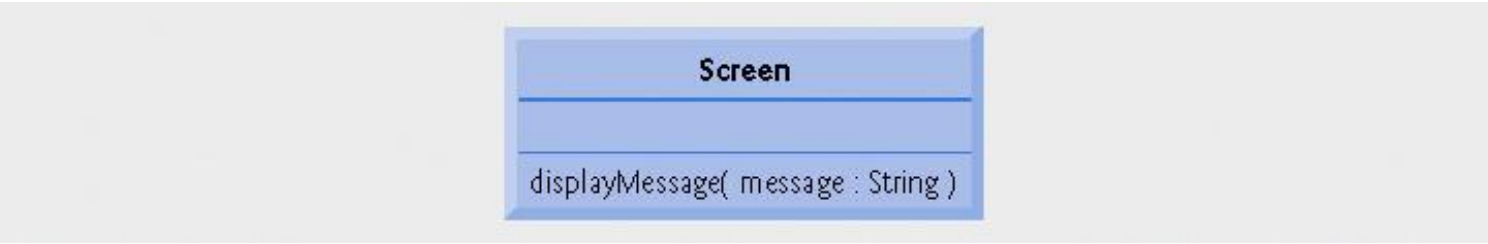


Figura 6.24 | Classe `Screen` com parâmetros de operação.

Figura 6.25 | Classe `CashDispenser` com parâmetros de operação.