5

Instruções de controle: Parte 2

OBJETIVOS

- Neste capítulo, você aprenderá:
- Os princípios básicos da repetição controlada por contador.
- Como utilizar as instruções de repetição `for` e `do…while` para executar instruções em um programa repetidamente.
- Como entender a seleção múltipla utilizando a instrução de seleção `switch`.
- Como utilizar as instruções `break` e `continue` para alterar o fluxo de controle.
- Como utilizar os operadores lógicos para formar expressões condicionais complexas em instruções de controle.

5.1 Introdução

- **Continuação da discussão sobre programação estruturada:**
  - **Introduz as demais instruções de controle do Java:**
    - `for`, `do…while` e `switch`.

5.2 Princípios básicos de repetição controlada por contador

- **A realização da repetição controlada por contador requer:**
  - **uma *variável de controle* (contador de loop);**
  - **o *valor inicial* da variável de controle;**
  - **o *incremento/decremento* da variável de controle a cada passagem do loop; e**
  - **a *condição de continuação do loop*, que determina se o loop deve continuar.**

Resumo

WhileCounter.java

```
1  // Fig. 5.1: WhileCounter.java
2  // Repetição controlada por contador com a instrução de repetição while.
3
4  public class WhileCounter
5  {
6     public static void main( String args[] )
7     {
8        int counter = 1; // declara e inicializa a variável de controle
9
10       while ( counter <= 10 ) // condição
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // incrementa a variáv
14       } // fim do while
15
16       System.out.println(); // gera a s                nha
17    } // fim de main
18 } // fim da classe WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

O nome da variável de controle é `counter`

O valor inicial da variável de controle é `1`

A condição testa o valor final de `counter`

Incremento para `counter`

Erro comum de programação 5.1

Uma vez que valores de ponto flutuante podem ser aproximados, controlar loops com variáveis de ponto flutuantes pode resultar em valores de contador e testes de terminação imprecisos.

Dica de prevenção de erro 5.1

Controle a contagem de loops com inteiros.

Boa prática de programação 5.1

Coloque linhas em branco acima e abaixo das instruções de controle de repetição e seleção e recue os corpos da instrução para aprimorar a legibilidade.

Observação de engenharia de software 5.1

‘Manter a coisa simples’ permanece um bom conselho para a maior parte do código que você escreverá.

5.3 A instrução de repetição for

- **Trata dos detalhes da repetição controlada por contador.**

Resumo

```
1  // Fig. 5.2: ForCounter.java
2  // Repetição controlada por contador com a instrução de repetição for.
3
4  public class ForCounter
5  {
6     public static void main( String args[] )
7     {
8        // cabeçalho da instrução for inclui inicialização,
9        // condição de continuação do loop e incremento
10       for ( int counter = 1; counter <= 10; counter++ )
11          System.out.printf( "%d  ", counter );
12
13       System.out.println(); // gera a saída de um caractere de nova linha
14    } // fim do main
15 }
```

O nome da variável de controle é `counter`

O valor inicial da variável de controle é `1`

Incremento para `counter`

A condição testa o valor final de `counter`

ForCounter.java

Linha 10
`int counter = 1;`

Linha 10
`int counter = 10;`

Erro comum de programação 5.2

Utilizar um operador relacional incorreto ou um valor final incorreto de um contador de loop na condição de continuação do loop de uma instrução de repetição pode causar um erro *off-by-one* (erro por um).

Boa prática de programação 5.2

Utilizar o valor final na condição de uma instrução `while` ou `for` e utilizar o operador relacional `<=` ajuda a evitar erros de *off-by-one*. Para um loop que imprime os valores de 1 a 10, a condição de continuação do loop deve ser `counter <= 10` em vez de `counter < 10` (o que causa um erro de *off-by-one*) ou `counter <= 11` (que é o correto).

Muitos programadores preferem a chamada contagem baseada em zero, na qual, para contar 10 vezes, `counter` seria inicializado como zero e o teste de continuação do loop seria `counter < 10`.

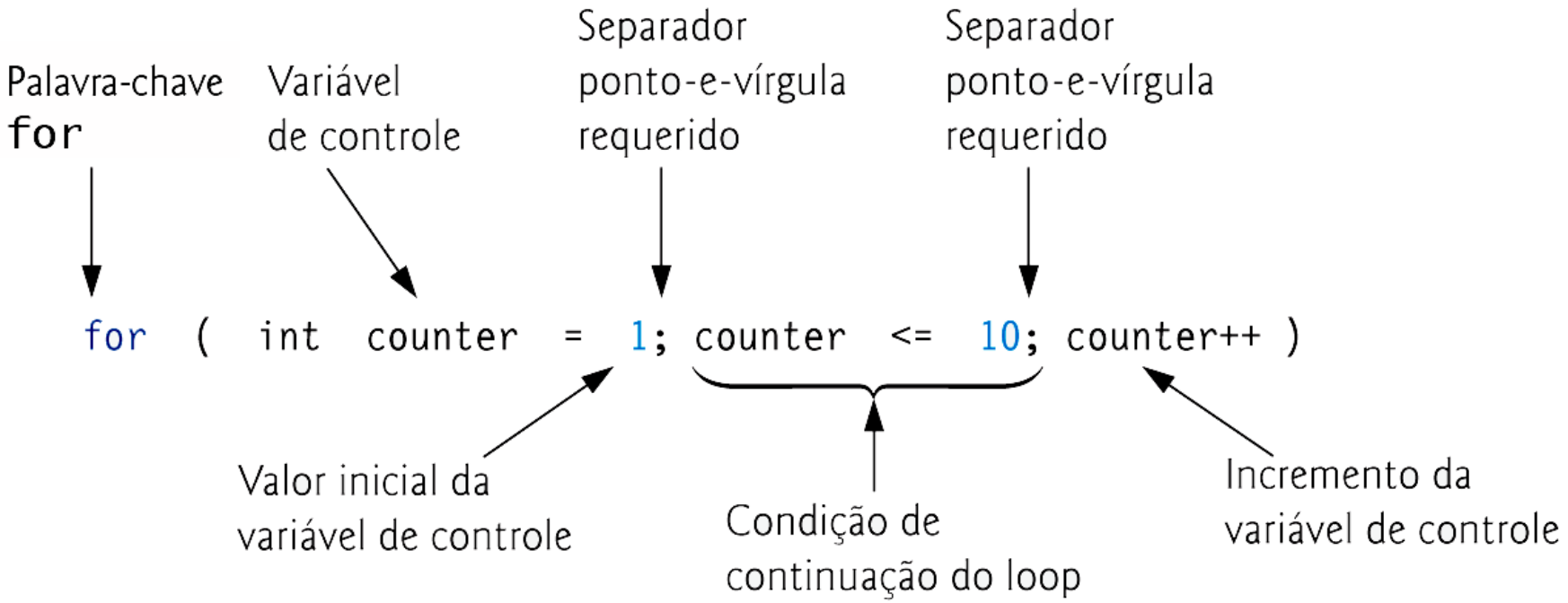


Figura 5.3 | Componentes do cabeçalho de uma instrução `for`.

5.3 A estrutura de repetição `for` (*Continuação*)

- `for` (*inicialização*; *condiçãoDeContinuaçãoDoLoop*; *incremento*)
  instrução;

- normalmente pode ser reescrito como:

- *inicialização*;
  `while` (*condiçãoDeContinuaçãoDoLoop*);
  `{`
  instrução;
  incremento;
  }

Erro comum de programação 5.3

Utilizar vírgulas em vez dos obrigatórios dois pontos-e-vírgulas em um cabeçalho `for` é um erro de sintaxe.

Erro comum de programação 5.4

Quando a variável de controle de uma instrução `for` for declarada na seção de inicialização do cabeçalho `for`, utilizar a variável de controle depois do corpo do `for` é um erro de compilação.

Dica de desempenho 5.1

Há uma ligeira vantagem de desempenho em pré-incrementar, mas, caso você escolha pós-incrementar porque parece mais natural (como em um cabeçalho `for`), otimizar os compiladores irá gerar o bytecode Java que, de qualquer maneira, utiliza a forma mais eficiente.

Boa prática de programação 5.3

Na maioria dos casos, tanto a pré-incrementação como a pós-incrementação são utilizadas para adicionar 1 a uma variável em uma instrução por si mesma. Nesses casos, o efeito é exatamente o mesmo, exceto que pré-incrementar tem uma ligeira vantagem de desempenho.

Dado que o compilador em geral otimiza seu código para ajudá-lo a obter o melhor desempenho, utilize o idioma com o qual você se sente mais à vontade nessas situações.

Erro comum de programação 5.5

Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito do cabeçalho de um `for` torna o corpo desse `for` uma instrução vazia.

Normalmente, esse é um erro de lógica.

Dica de prevenção de erro 5.2

Os loops infinitos ocorrem quando a condição de continuação do loop em uma instrução de repetição não se torna `false`. Para evitar essa situação em um loop controlado por contador, assegure que a variável de controle é incrementada (ou decrementada) a cada iteração do loop.

Em um loop controlado por sentinela, certifique-se de que o valor da sentinela seja por fim inserido.

Dica de prevenção de erro 5.3

Embora o valor da variável de controle possa ser alterado no corpo de um loop `for`, evite fazê-lo assim porque essa prática pode levar a erros sutis.

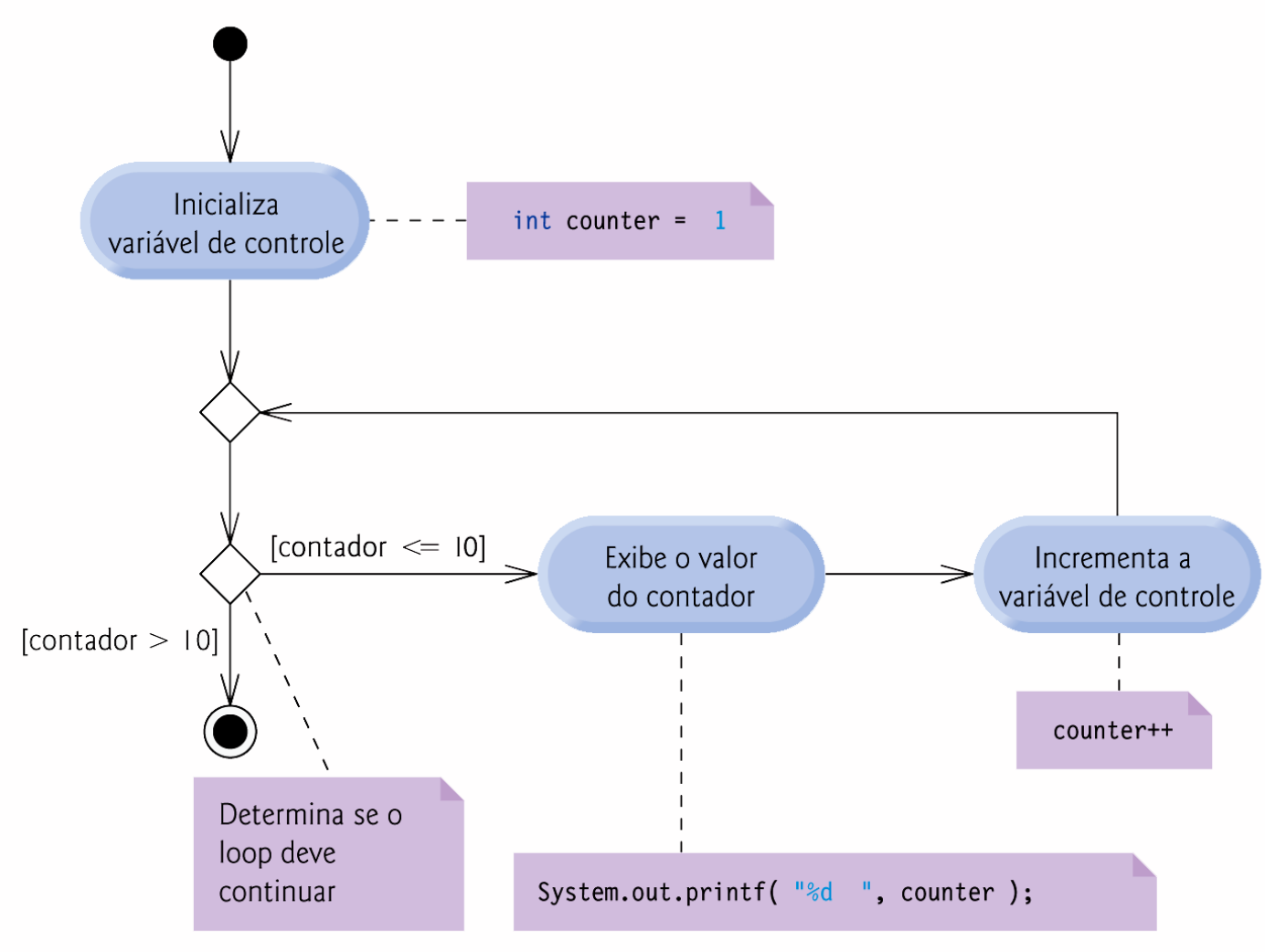


Figura 5.4 | Diagrama de atividade UML para a instrução `for` na Figura 5.2.

5.4 Exemplos com a estrutura for

- **Variando a variável de controle em uma instrução for**
  - **Faz a variável de controle variar de 1 a 100 em incrementos de 1**
    - `for ( int i = 1; i <= 100; i++ )`
  - **Faz a variável de controle variar de 100 a 1 em incrementos de −1**
    - `for ( int i = 100; i >= 1; i-- )`
  - **Faz a variável de controle variar de 7 a 77 em incrementos de 7**
    - `for ( int i = 7; i <= 77; i += 7 )`
  - **Faz a variável de controle variar de 20 a 2 em decrementos de 2**
    - `for ( int i = 20; i >= 2; i -= 2 )`
  - **Faz a variável de controle variar na seqüência: 2, 5, 8, 11, 14, 17, 20**
    - `for ( int i = 2; i <= 20; i += 3 )`
  - **Faz a variável de controle variar na seqüência : 99, 88, 77, 66, 55, 44, 33, 22, 11, 0**
    - `for ( int i = 99; i >= 0; i -= 11 )`

Erro comum de programação 5.6

Não utilizar o operador relacional adequado na condição de continuação de um loop que conta para baixo (por exemplo, utilizar `i <= 1` em vez de `i >= 1` em uma contagem de loop para baixo até 1) normalmente é um erro de lógica.

Resumo

Sum.java

Linha 11

```
1  // Fig. 5.5: Sum.java
2  // Somando inteiros com a instrução for.
3
4  public class Sum
5  {
6     public static void main( String args[] )
7     {
8        int total = 0; // inicializa o total
9
10       // total de inteiros pares de 2 a 20
11       for ( int number = 2; number <= 20; number += 2 )
12          total += number;
13
14       System.out.printf( "Sum is %d\n", total ); // exibe os resultados
15    } // fim de main
16 } // fim da classe Sum
```

```
Sum is 110
```

incrementa `number` por 2 a cada iteração

5.4 Exemplos com a estrutura for (*Continuação*)

- **As expressões *inicialização* e *incremento* podem ser listas separadas por vírgulas de expressões que permitem ao programador utilizar múltiplas expressões de inicialização ou múltiplas expressões de incremento.**
  - **Por exemplo, as linhas 11-12 da Figura 5.5 podem ser reescritas como:**
  -
```
for ( int number = 2; number <= 20; total +=
number, number += 2 )
    ;  // instrução vazia
```

Boa prática de programação 5.4

Limite o tamanho de cabeçalhos de instrução de controle a uma única linha, se possível.

Boa prática de programação 5.5

Nas seções de inicialização e incremento de uma estrutura `for` coloque apenas expressões que envolvem as variáveis de controle.

As manipulações de outras variáveis devem aparecer antes do loop (se executarem apenas uma vez, como as instruções de inicialização) ou no corpo do loop (se executarem uma vez por iteração do loop, como as instruções de incremento ou decremento).

Resumo

Interest.java

(1 de 2)

Linha 8

Linha 13

```
1  // Fig. 5.6: Interest.java
2  // Cálculos de juros compostos com for.
3
4  public class Interest
5  {
6     public static void main( String args[
7     {
8        double amount; // quantia em depósito ao fim de cada ano
9        double principal = 1000.0; // quantia inicial antes dos juros
10       double rate = 0.05; // taxa de juros
11
12       // exibe cabeçalhos
13       System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
14
```

O Java trata pontos flutuantes como tipo `double`

A segunda string é justificada à direita e exibida com uma largura de campo de 20

Resumo

Interest.java

(2 de 2)

```
15       // calcula quantidade de depósito para cada um dos dez anos
16       for ( int year = 1; year <= 10; year++ )
17       {
18          // calcula nova quantidade durante ano especificado
19          amount = principal * Math.pow( 1.0 + rate, year );
20
21          // exibe o ano e a quantidade
22          System.out.printf( "%4d%,20.2f\n", year, amount );
23       } // fim do for
24    } // fim de main
25 } // fim da classe Interest
```

```
Year    Amount on deposit
   1            1,050.00
   2            1,102.50
   3            1,157.63
   4            1,215.51
   5            1,276.28
   6            1,340.10
   7            1,407.10
   8            1,477.46
   9            1,551.33
  10            1,628.89
```

Calcule `amount` com uma instrução `for`

Utilize o flag de formatação vírgula (`,`) para exibir o valor com um separador de milhares

Linhas 16-23

Linha 22

Saída do programa

5.4 Exemplos com a estrutura `for` (*Continuação*)

- **Formatando a saída:**
  - **Largura do campo.**
  - **Flag de formatação sinal de subtração (`-`).**
  - **Flag de formatação vírgula (`,`).**
- **Método `static`**
  - ***NomeDaClasse.nomeDoMétodo(argumentos)***

Boa prática de programação 5.6

Não utilizar variáveis de tipo `double` (ou `float`) para realizar cálculos monetários precisos. A imprecisão de números de ponto flutuante pode causar erros que resultarão em valores monetários incorretos. Nos exercícios, exploramos o uso de inteiros para realizar cálculos monetários. [Nota: Alguns fornecedores independentes comercializam bibliotecas de classe que realizam cálculos monetários precisos. Além disso, a API do Java fornece a classe `java.math.BigDecimal` para realizar cálculos com valores de ponto flutuante de precisão arbitrária.]

Dica de desempenho 5.2

Em loops, evite cálculos para os quais o resultado nunca muda — esses cálculos em geral devem ser colocados antes do loop.

[Nota: Muitos compiladores de otimização sofisticados de hoje colocarão esses cálculos fora de loops no código compilado.]

5.5 A estrutura de repetição do…while

- **Estrutura `do...while`:**
  - **Semelhante à instrução `while`.**
  - **Testa a continuação do loop depois de executar o corpo do loop.**
    - **Isto é, o corpo do loop sempre é executado pelo menos uma vez.**

Resumo

```
1  // Fig. 5.7: DoWhileTest.java
2  // Instrução de repetição do...while.
3
4  public class DoWhileTest
5  {
6     public static void main( String args[] )
7     {
8        int counter = 1; // inicializa cou
9
10       do
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter;
14       } while ( counter <= 10 ); // fim do do...while
15
16       System.out.println(); // gera a saída de um caractere de nova linha
17    } // fim de main
18 } // fim da classe DoWhileTest
```

Declara e inicializa o contador da variável `counter`

Valor da variável `counter` é exibido antes do teste do valor final de `counter`

```
1  2  3  4  5  6  7  8  9  10
```

DoWhileTest.java

Linha 8

Linhas 10-14

Saída do programa

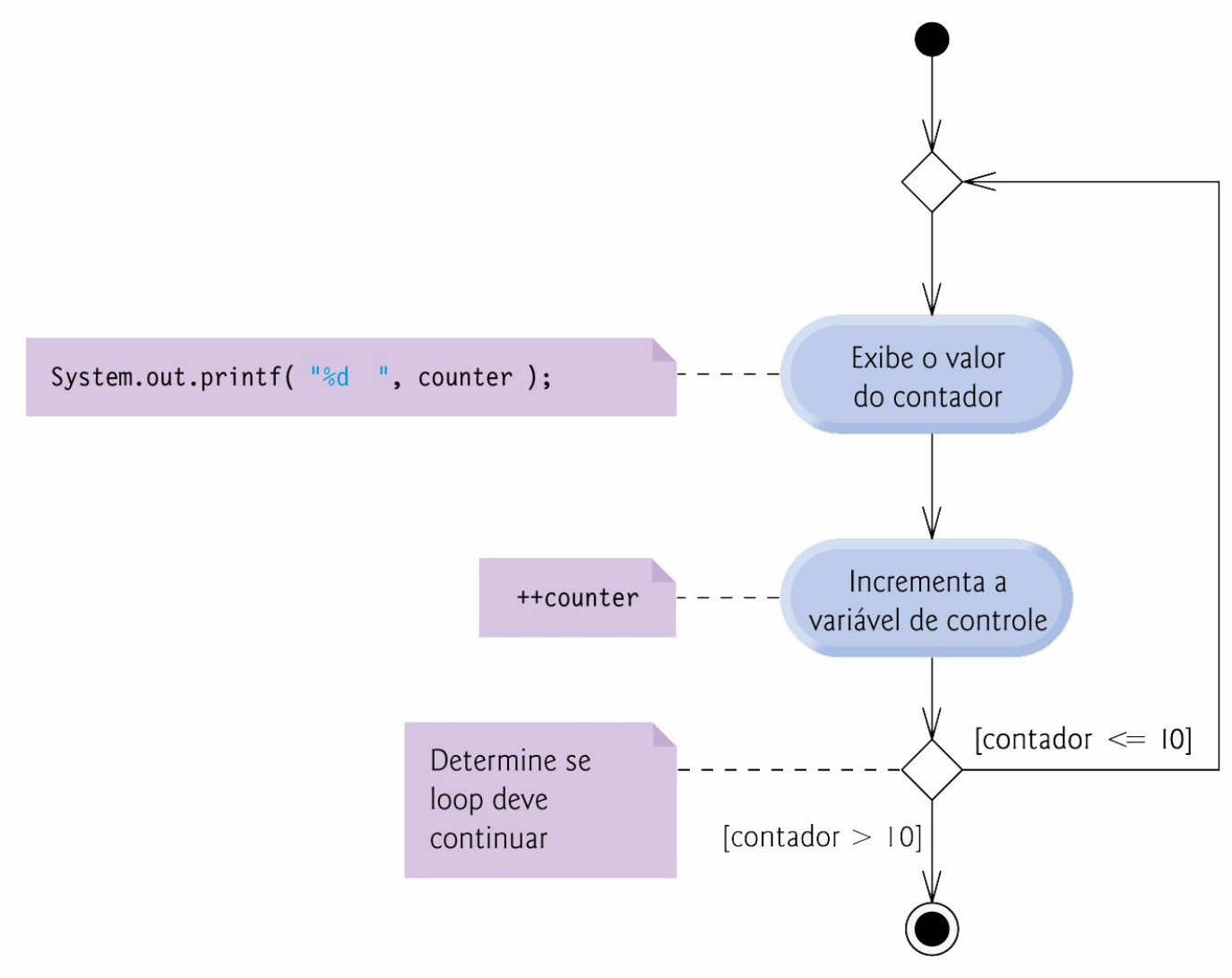


Figura 5.8 | Diagrama de atividades UML de uma instrução de repetição `do...while`.

Boa prática de programação 5.7

Sempre inclua chaves em uma instrução `do...while`, mesmo se elas não forem necessárias.

Isso ajuda a eliminar ambigüidade entre a instrução `while` e uma instrução `do...while` que contém somente uma instrução.

5.6 A estrutura de seleção múltipla `switch`

- **Instrução `switch`:**
  - **Utilizada para múltiplas seleções.**

Resumo

GradeBook.java

(1 de 5)

Linhas 8-14

```
1  // Fig. 5.9: GradeBook.java
2  // Classe GradeBook usa instrução switch para contar as notas A, B, C, D e F..
3  import java.util.Scanner; // programa utiliza a classe Scanner
4
5  public class GradeBook
6  {
7     private String courseName; // nome do curso que essa GradeBook representa
8     private int total; // soma das notas
9     private int gradeCounter; // número de notas inseridas
10    private int aCount; // contagem de notas A
11    private int bCount; // contagem de notas B
12    private int cCount; // contagem de notas C
13    private int dCount; // contagem de notas D
14    private int fCount; // contagem de notas F
15
16    // construtor inicializa courseName;
17    // variáveis de instância int são inicializadas como 0 por padrão
18    public GradeBook( String name )
19    {
20       courseName = name; // inicializa courseName
21    } // fim do construtor
22
23    // método para configurar o nome do curso
24    public void setCourseName( String name )
25    {
26       courseName = name; // armazena o nome do curso
27    } // fim do método setCourseName
28
```

Resumo

GradeBook.java

(2 de 5)

Linhas 50-54

```
29    // método para recuperar o nome do curso
30    public String getCourseName()
31    {
32       return courseName;
33    } // fim do método getCourseName
34
35    // exibe uma mensagem de boas-vindas para o usuário GradeBook
36    public void displayMessage()
37    {
38       // getCourseName obtém o nome do curso
39       System.out.printf( "Welcome to the grade book for\n%s!\n\n",
40          getCourseName() );
41    } // fim do método displayMessage
42
43    // insere número arbitrário de notas do usuário
44    public void inputGrades()
45    {
46       Scanner input = new Scanner( System.in );
47
48       int grade; // nota inserida pelo usuário
49
50       System.out.printf( "%s\n%s\n   %s\n   %s\n",
51          "Enter the integer grades in the range 0-100.",
52          "Type the end-of-file indicator to terminate input:",
53          "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
54          "On Windows type <ctrl> z then press Enter" );
55
```

Exibe o prompt

Resumo

GradeBook.java

(3 de 5)

Linha 57

Linha 72 expressão de controle

Linhas 72-94

```
56      // faz loop até o usuário inserir o indicador de fim do arquivo
57      while ( input.hasNext() )
58      {
59         grade = input.nextInt();// lê a n
60         total += grade; // adiciona grade
61         ++gradeCounter; // incrementa o c
62
63         // chama método para incrementar o contador adequado
64         incrementLetterGradeCounter( grade );
65      } // fim do while
66   } // fim do método inputGrades
67
68   // adiciona 1 ao contador adequado da nota especificada
69   public void incrementLetterGradeCounter( int numericGrade )
70   {
71      // determina a nota que foi inserida
72      switch ( grade / 10 )
73      {
74         case 9:  // nota estava entre 90
75         case 10: // e 100
76            ++aCount; // incrementa aCount
77            break; // necessário para sair
78
79         case 8: // nota estava entre 80 e 89
80            ++bCount; // incrementa bCount
81            break; // sai do switch
82
```

A condição de loop utiliza o método `hasNext` para determinar se há mais dados a inserir

(`grade / 10`) é *a expressão de controle*

A instrução `switch` determina qual rótulo `case` executar, dependendo da expressão de controle

```
83          case 7: // nota estava entre 70 e 79
84             ++cCount; // incrementa cCount
85             break; // sai do switch
86
87          case 6: // nota estava entre 60 and 69
88             ++dCount; // incrementa dCount
89             break; // exit switch
90
91          default: // nota era menor que 60
92             ++fCount; // incrementa fCount
93             break; // opcional; sairá de switch de qualquer jeito
94       } // fim do switch
95    } // fim do método inc
96
97    // exibe um relatório baseado nas notas inseridas pelo usuário
98    public void displayGradeReport()
99    {
100      System.out.println( "\nGrade Report:" );
101
102      // se usuário inseriu pelo menos uma nota...
103      if ( gradeCounter != 0 )
104      {
105         // calcula a média de todas as notas inseridas
106         double average = (double) total / gradeCounter;
107
```

`default case` para notas menores que 60

Resumo

`GradeBook.java`

(4 de 5)

Linha 91 caso-padrão

Resumo

GradeBook.java

(5 de 5)

```
108         // gera a saída de resumo dos resultados
109         System.out.printf( "Total of the %d grades entered is %d\n",
110            gradeCounter, total );
111         System.out.printf( "Class average is %.2f\n", average );
112         System.out.printf( "%s\n%s%d\n%s%d\n%s%d\n%s%d\n%s%d\n",
113            "Number of students who received each grade:",
114            "A: ", aCount,   // exibe número de notas A
115            "B: ", bCount,   // exibe número de notas B
116            "C: ", cCount,   // exibe número de notas C
117            "D: ", dCount,   // exibe número de notas D
118            "F: ", fCount ); // exibe número de notas F
119      } // fim do if
120      else // nenhuma nota inserida, assim gera a saída da mensagem apropriada
121         System.out.println( "No grades were entered" );
122   } // fim do método displayGradeReport
123} // fim da classe GradeBook
```

Dica de portabilidade 5.1

As combinações de teclas pressionadas para inserir o fim do arquivo são dependentes de sistema.

Erro comum de programação 5.7

Esquecer uma instrução `break` quando essa for necessária em uma estrutura `switch` é um erro de lógica.

Resumo

```
1  // Fig. 5.10: GradeBookTest.java
2  // Cria o objeto GradeBook, insere notas e exibe relatório de notas.
3
4  public class GradeBookTest
5  {
6     public static void main( String args[] )
7     {
8        // cria o objeto myGradeBookda classe GradeBook e
9        // passa o nome de cursor para o construtor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
14       myGradeBook.inputGrades(); // lê as notas fornecidas pelo usuário
15       myGradeBook.displayGradeReport(); // exibe relatório com base nas notas
16    } // fim de main
17 } // fim da classe GradeBookTest
```

GradeBookTest.java

Chama os métodos `public` `GradeBook` para contar as notas

Linhas 13-15

```
Welcome to the grade book for
CS101 Introduction to Java Programming!

Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/Mac OS X type <ctrl> d then press Enter
   On Windows type <ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Resumo

GradeBookTest.java

(2 de 2)

Saída do programa

Observação de engenharia de software 5.2

Forneça um caso `default` em instruções `switch`.

Incluir um caso `default` faz com que você se concentre na necessidade de processar condições excepcionais.

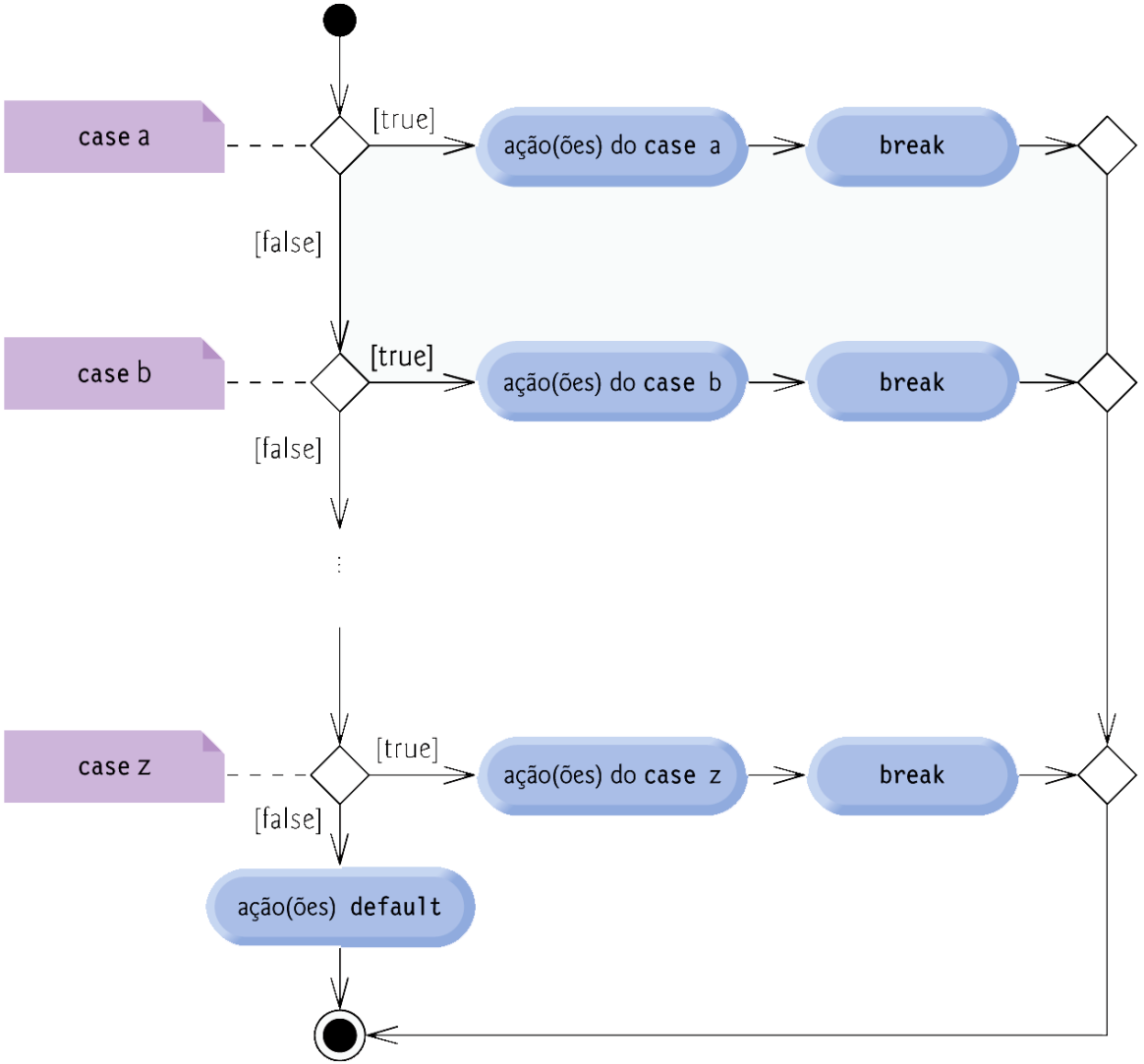


Figura 5.11 | Diagrama de atividade UML de instrução de seleção múltipla `switch` com instruções `break`.

Boa prática de programação 5.8

Embora cada `case` e a instrução `default` em uma `switch` possam ocorrer em qualquer ordem, coloque a instrução `default` por último.

Quando a instrução `default` é listada por último, o `break` para essa instrução não é necessário.

Alguns programadores incluem esse `break` para clareza e simetria com outros `case`s.

5.6 A estrutura de seleção múltipla `switch` (*Continuação*)

- **Expressão em cada `case`:**
  - **Expressão integral constante:**
    - **Combinação de constantes inteiras que são avaliadas como um valor inteiro constante.**
  - **Caractere constante:**
    - **Por exemplo, `'A'`, `'7'` ou `'$'`.**
  - **Variável constante:**
    - **Declarado com a palavra-chave `final`.**

5.7 Instruções break e continue

- break/continue:
  - Altera o fluxo de controle.
- Instrução break:
  - Causa saída imediata da estrutura de controle.
    - Utilizada nas instruções while, for, do…while ou switch.
- Instrução continue:
  - Pula as instruções restantes no corpo do loop.
  - Prossegue para a próxima iteração.
    - Utilizada nas instruções while, for, do…while ou switch.

Resumo

```
1  // Fig. 5.12: BreakTest.java
2  // a instrução break sai de uma instrução for.
3  public class BreakTest
4  {
5     public static void main( String args[] )
6     {
7        int count; // variável de controle também
8
9        for ( count = 1; count <= 10; count++ ) //
10       {
11          if ( count == 5 ) // se count for 5,
12             break;         // termina o loop
13
14          System.out.printf( "%d ", count );
15       } // fim do for
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // fim do main
19 } // fim da classe BreakTest
```

```
1 2 3 4
Broke out of loop at count = 5
```

Executa o loop 10 vezes

Saída para a instrução `for` (`break`) quando a contagem é igual a 5

`Test.java`

`Linha 9`

`Linhas 11-12`

`Saída do programa`

Resumo

```
1  // Fig. 5.13: ContinueTest.java
2  // instrução continue que termina a iteração de uma instrução for.
3  public class ContinueTest
4  {
5     public static void main( String args[] )
6     {
7        for ( int count = 1; count <= 10; count++ )
8        {
9           if ( count == 5 ) // se count for 5,
10             continue;      // pula o restante do código no loop
11
12          System.out.printf( "%d ", count );
13       } // fim do for
14
15       System.out.println( "\nUsed continue to skip printing 5" );
16    } // fim do main
17 } // fim da classe ContinueTest
```

Executa o loop 10 vezes

Pula a linha 12 e prossegue para a linha 7 quando count é igual a 5

Linha 7

Linhas 9-10

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Saída do programa

Observação de engenharia de software 5.3

Alguns programadores consideram que `break` **e** `continue` **violam a programação estruturada.**

Visto que os mesmos efeitos são alcançáveis com as técnicas de programação estruturada, esses programadores não utilizam `break` **ou** `continue`**.**

Observação de engenharia de software 5.4

Há uma tensão entre alcançar uma engenharia de software de qualidade e alcançar o software de melhor desempenho.

Freqüentemente, um desses objetivos é alcançado à custa do outro. Para todas as situações — exceto as de desempenho muito alto —, aplique a seguinte regra geral: primeiro, faça seu código simples e correto; então, torne-o rápido e pequeno, mas apenas se necessário.

5.8 Operadores lógicos

- **Operadores lógicos:**
  - **Permite formar condições mais complexas.**
  - **Combina condições simples.**
- **Operadores lógicos Java:**
  - **&&** **(E condicional)**
  - **||** **(OU condicional)**
  - **&** **(E lógico booleano)**
  - **|** **(OU inclusivo lógico booleano)**
  - **^** **(OU exclusivo lógico booleano)**
  - **!** **(NÃO lógico)**

5.8 Operadores lógicos (*Continuação*)

- **Operador E (&&) condicional.**
  - **Considere a seguinte instrução `if`:**
    - `if ( gender == FEMALE && age >= 65 )`
    - `++++seniorFemales;`
    - **Condição combinada é `true`:**
    - **se e somente se ambas as condições simples forem `true`.**
    - **Condição combinada é `false`:**
    - **se uma ou ambas as condições simples forem `false`.**

expressão1	expressão2	expressão1 && expressão2
false	false	False
false	true	False
true	false	False
true	true	True

Figura 5.14 | Tabela-verdade do operador && (E condicional).

5.8 Operadores lógicos (*Continuação*)

- **Operador OU condicional (||):**
  - **Considere a seguinte instrução `if`:**
    - `if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )`
    - `System.out.println ( “Student grade is A” );`
    - **A condição combinada é `true`:**
    - **se uma das ou ambas as condições simples forem `true`.**
    - **A condição combinada é `false`:**
    - **se ambas as condições simples forem `false`.**

expressão1	expressão2	expressão1 || expressão2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 5.15 | Tabela-verdade do operador || (OU condicional).

5.8 Operadores lógicos (*Continuação*)

- **Avaliação de curto-circuito de condições complexas:**
  - **As partes de uma expressão contendo os operadores && ou || só são avaliadas quando se sabe se a condição é verdadeira ou falsa.**
  - **Por exemplo:**
  - `( gender == FEMALE ) && ( age >= 65 )`
    - **Pára imediatamente se o sexo não `for` igual a `FEMALE`.**

Erro comum de programação 5.8

Em expressões que utilizam o operador &&, uma condição — que chamaremos de condição dependente — pode exigir que outra condição seja verdadeira para que a avaliação da condição dependente tenha significado. Neste caso, a condição dependente deve ser colocada depois da outra condição, ou um erro pode ocorrer.

Por exemplo, na expressão `( i != 0 ) && ( 10 / i == 2 )`, a segunda condição deve aparecer depois da primeira condição ou um erro de divisão por zero pode ocorrer.

5.8 Operadores lógicos (*Continuação*)

- **Operador lógico booleano E (&):**
  - **Funciona de maneira idêntica a &&.**
  - **Exceto que & sempre avalia ambos os operandos.**
- **Operador booleano lógico OU (|):**
  - **Funciona de maneira idêntica a ||.**
  - **Exceto que | sempre avalia ambos os operandos.**

Dica de prevenção de erro 5.4

Por questão de clareza, evite expressões com efeitos colaterais nas condições.

Os efeitos colaterais talvez pareçam inteligentes, mas podem dificultar o entendimento do código e podem levar a erros de lógica sutis.

5.8 Operadores lógicos (*Continuação*)

- **OU exclusivo lógico booleano (^):**
  - **Um dos seus operandos é `true` e o outro é `false`:**
    - **Avaliado como `true`.**
  - **Ambos os operandos são `true` ou ambos são `false`:**
    - **Avaliado como `false`.**
- **Operador de negação lógica (`!`):**
  - **Operador unário.**

expressão1	expressão2	expressão1 ^ expressão2
false	false	false
false	true	true
true	false	true
true	true	false

Figura 5.16 | Tabela-verdade do operador ^ (OU exclusivo lógico booleano).

expressão	!expressão
`false`	`true`
`true`	`false`

Figura 5.17 | Tabela-verdade do operador ! (negação lógica ou NÃO lógico).

Resumo

LogicalOperators.java

(1 de 3)

Linhas 9-13

Tabela-verdade do E condicional

Linhas 16-20

Tabela-verdade do OU condicional

Linhas 23-27

Tabela-verdade do E lógico booleano

```
1  // Fig. 5.18: LogicalOperators.java
2  // Operadores lógicos.
3
4  public class LogicalOperators
5  {
6     public static void main( String args[] )
7     {
8        // cria a tabela-verdade para o operador && (E condicional)
9        System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
10          "Conditional AND (&&)", "false && false", ( false && false ),
11          "false && true", ( false && true ),
12          "true && false", ( true && false ),
13          "true && true", ( true && true ) );
14
15       // cria a tabela-verdade para o operador || (OU condicional)
16       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
17          "Conditional OR (||)", "false || false", ( false || false ),
18          "false || true", ( false || true ),
19          "true || false", ( true || false ),
20          "true || true", ( true || true ) );
21
22       // cria a tabela-verdade para o operador & (E lógico booleano)
23       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
24          "Boolean logical AND (&)", "false & false", ( false & false ),
25          "false & true", ( false & true ),
26          "true & false", ( true & false ),
27          "true & true", ( true & true ) );
28
```

Resumo

```
29       // cria a tabela-verdade para | (OU inclusivo lógico booleano) operador
30       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
31          "Boolean logical inclusive OR (|)",
32          "false | false", ( false | false ),
33          "false | true", ( false | true ),
34          "true | false", ( true | false ),
35          "true | true", ( true | true ) );
36
37       // cria a tabela-verdade para o operador ^ (OU exclusivo lógico booleano)
38       System.out.printf( "%s\n%s: %b\n%s: %b\n%s: %b\n%s: %b\n\n",
39          "Boolean logical exclusive OR (^)",
40          "false ^ false", ( false ^ false ),
41          "false ^ true", ( false ^ true ),
42          "true ^ false", ( true ^ false ),
43          "true ^ true", ( true ^ true ) );
44
45       // cria a tabela-verdade para o operador ! (negação lógica)
46       System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
47          "!false", ( !false ), "!true", ( !true ) );
48    } // fim de main
49 } // fim da classe LogicalOperators
```

Tabela-verdade do OU inclusivo lógico booleano

(2 de 3)

Linhas 30-35

Tabela-verdade do OU exclusivo lógico booleano

Linhas 46-47

Tabela-verdade da negação lógica

```
Conditional AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Conditional OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Boolean logical AND (&)
false & false: false
false & true: false
true & false: false
true & true: true

Boolean logical inclusive OR (|)
false | false: false
false | true: true
true | false: true
true | true: true

Boolean logical exclusive OR (^)
false ^ false: false
false ^ true: true
true ^ false: true
true ^ true: false

Logical NOT (!)
!false: true
!true: false
```

Resumo

LogicalOperators.java

(3 de 3)

Saída do programa

Operadores						Associatividade	Tipo
`++`	`--`					da direita para a esquerda	unário pós-fixo
`++`	`-`	`+`	`-`	`!`	`(`*tipo*`)`	da direita para a esquerda	unário pré-fixo
`*`	`/`	`%`				da esquerda para a direita	multiplicativo
`+`	`-`					da esquerda para a direita	aditivo
`<`	`<=`	`>`	`>=`			da esquerda para a direita	relacional
`==`	`!=`					da esquerda para a direita	igualdade
`&`						da esquerda para a direita	E lógico booleano
`^`						da esquerda para a direita	OU lógico booleano exclusivo
`|`						da esquerda para a direita	OU inclusivo lógico booleano
`&&`						da esquerda para a direita	E condicional
`||`						da esquerda para a direita	OU condicional
`?:`						da direita para a esquerda	ternário condicional
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	da direita para a esquerda	atribuição

Figura 5.19 | Precedência/associatividade dos operadores discutidos até agora.

5.9 Resumo de programação estruturada

- **Estrutura da seqüência:**
  - **‘predefinida’ no Java.**
- **Estrutura de seleção:**
  - `if`, `if…else` e `switch`.
- **Estrutura de repetição:**
  - `while`, `do…while` e `for`.

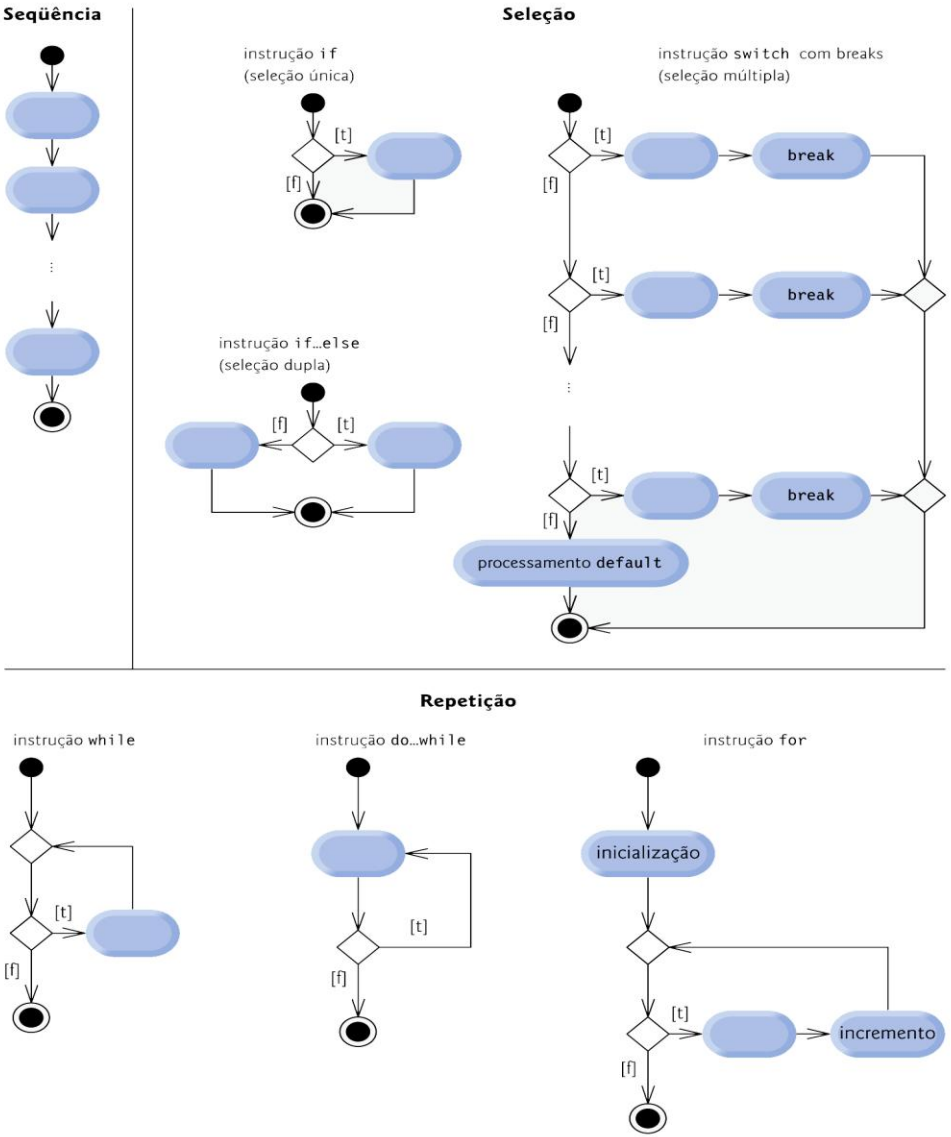


Figura 5.20 | Instruções de seqüência de entrada única/saída única, seleção e repetição do Java.

Lucrécia Freitas Ideal:

Checar corpo. As letras parecem condensadas demais.

Regras para formar programas estruturados	
1	Comece com o diagrama de atividade mais simples (Fig. 5.22).
2	Qualquer estado de ação pode ser substituído por dois estados de ação em seqüência.
3	Qualquer estado de ação pode ser substituído por qualquer instrução de controle (seqüência de estados de ação, `if`, `if else`, `switch`, `while`, `do while` ou `for`).
4	As regras 2 e 3 podem ser aplicadas com freqüência que você quiser em qualquer ordem.

Figura 5.21 | Regras para formar programas estruturados.

Figura 5.22 | Diagrama de atividades mais simples.

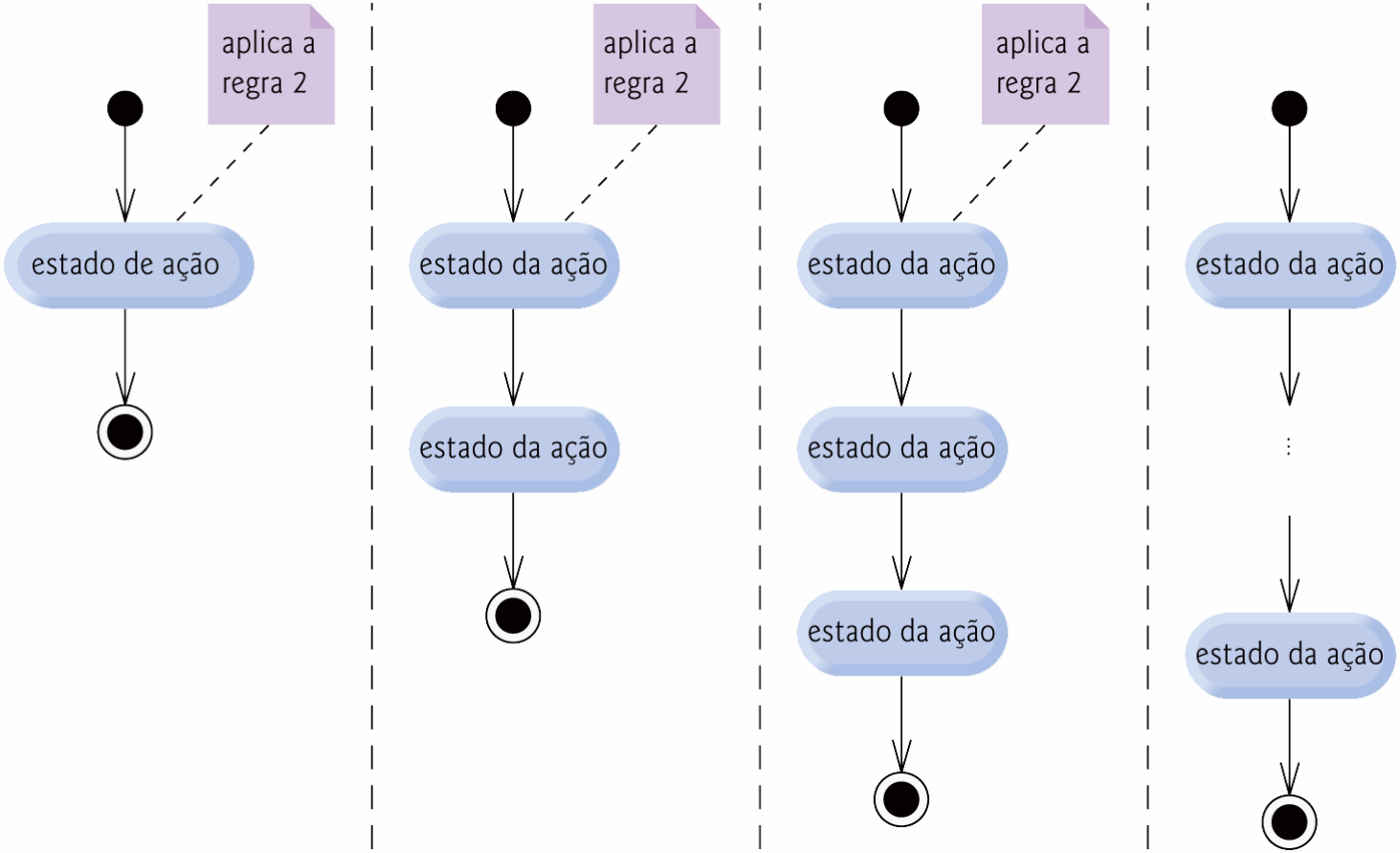


Figura 5.23 | Aplicando a regra de empilhamento repetidamente (regra 2) da Figura 5.21 ao diagrama de atividades mais simples.

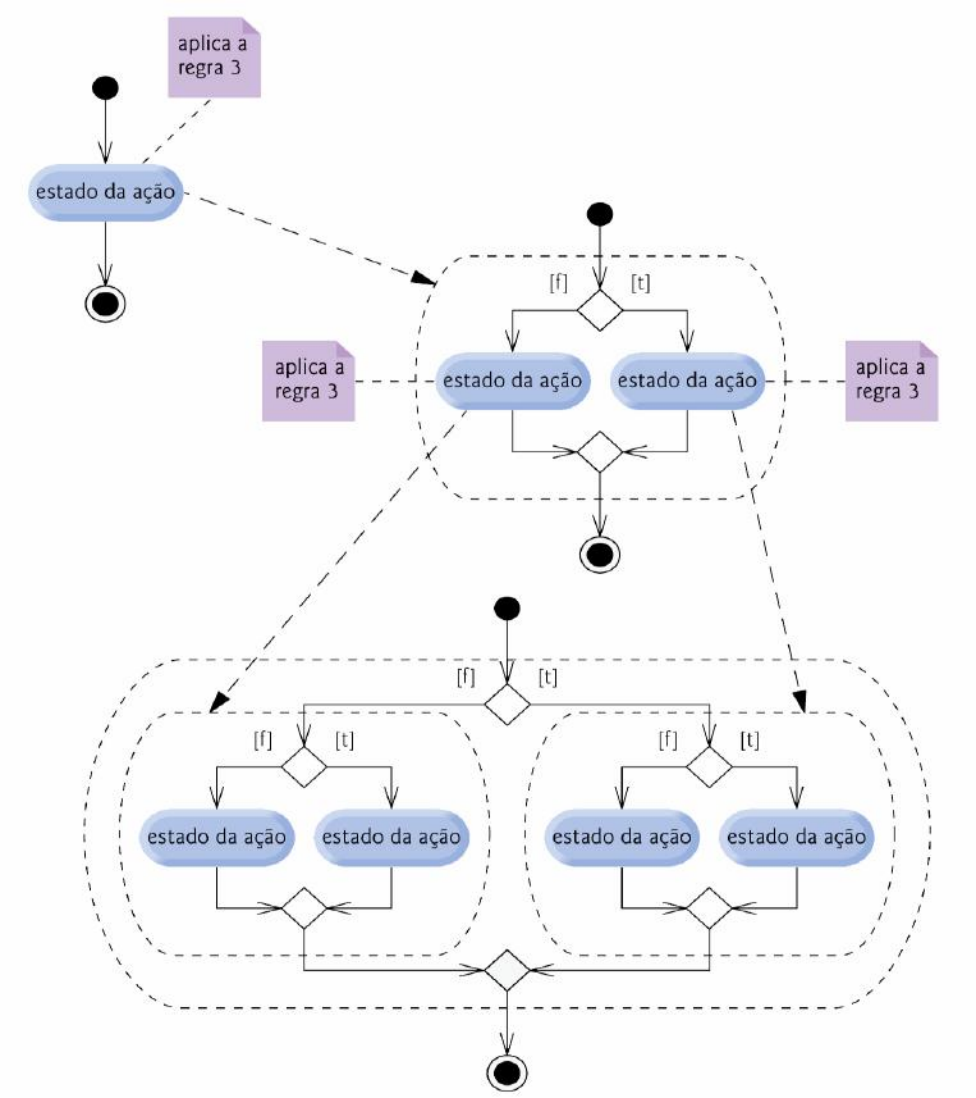


Figura 5.24 | Aplicando a regra de aninhamento repetidamente (regra 3) da Figura 5.21 ao diagrama de atividades mais simples.

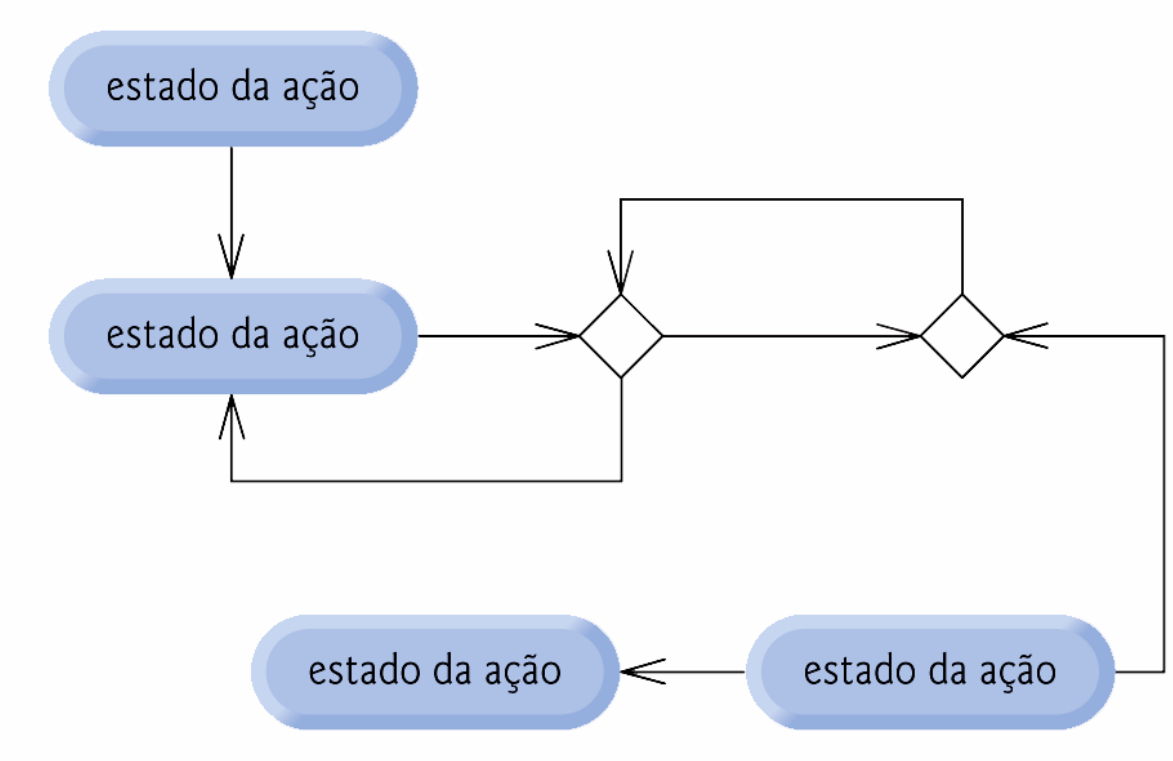


Figura 5.25 | Diagrama de atividades ‘não-estruturado’.

5.10 (Opcional) Estudo de caso de GUIs e imagens gráficas: Desenhando retângulos e ovais

- **Desenha retângulos:**
  - **Método `drawRect` de `Graphics`.**
- **Desenha ovais:**
  - **Método `drawOval` de `Graphics`.**

Resumo

Shapes.java

(1 de 2)

```
1  // Fig. 5.26: Shapes.java
2  // Demonstra o desenho de diferentes formas.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class Shapes extends JPanel
7  {
8     private int choice; // escolha do usuário de qual forma desenhar
9
10    // construtor configura a escolha do usuário
11    public Shapes( int userChoice )
12    {
13       choice = userChoice;
14    } // fim do construtor Shapes
15
16    // desenha uma cascata de formas que iniciam do canto superior esquerdo
17    public void paintComponent( Graphics g )
18    {
19       super.paintComponent( g );
20
```

```
21       for ( int i = 0; i < 10; i++ )
22       {
23          // seleciona a forma com base na escolha do usuário
24          switch ( choice )
25          {
26             case 1: // desenha retângulos
27                g.drawRect( 10 + i * 10, 10 + i * 10,
28                   50 + i * 10, 50 + i * 10 );
29                break;
30             case 2: // draw ovals
31                g.drawOval( 10 + i * 10, 10 + i * 10,
32                   50 + i * 10, 50 + i * 10 );
33                break;
34          } // fim do switch
35       } // fim do for
36    } // fim do método paintComponent
37 } // fim da classe Shapes
```

Desenha retângulos

Desenha ovais

Resumo

Shapes.java

(2 de 2)

Linhas 27-28

Linhas 31-32

Resumo

ShapesTest.java

(1 de 2)

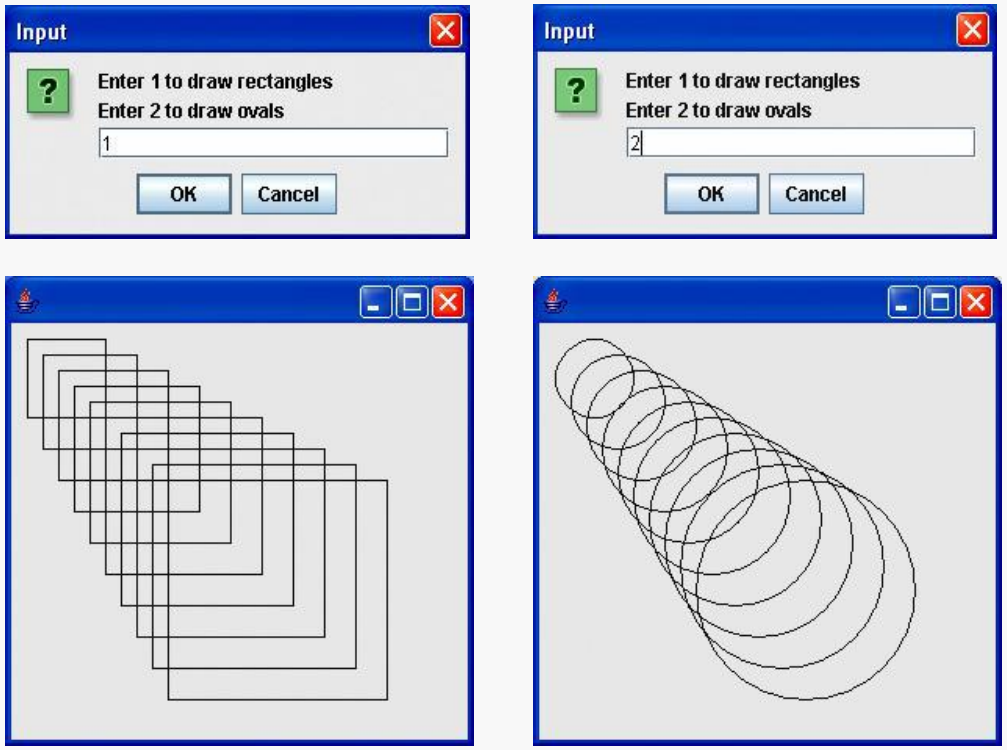
```
1  // Fig. 5.27: ShapesTest.java
2  // Aplicativo de teste que exibe a classe Shapes.
3  import javax.swing.JFrame;
4  import javax.swing.JOptionPane;
5
6  public class ShapesTest
7  {
8     public static void main( String args[] )
9     {
10       // obtém a escolha do usuário
11       String input = JOptionPane.showInputDialog(
12          "Enter 1 to draw rectangles\n" +
13          "Enter 2 to draw ovals" );
14
15       int choice = Integer.parseInt( input ); // convert input to int
16
17       // cria o painel com a entrada do usuário
18       Shapes panel = new Shapes( choice );
19
20       JFrame application = new JFrame(); // cria um novo JFrame
21
22       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
23       application.add( panel ); // adiciona o painel ao frame
24       application.setSize( 300, 300 ); // configura o tamanho desejado
25       application.setVisible( true ); // mostra o frame
26    } // fim de main
27 } // fim da classe ShapesTest
```

Resumo

`ShapesTest.java`

(2 de 2)

Saída do programa



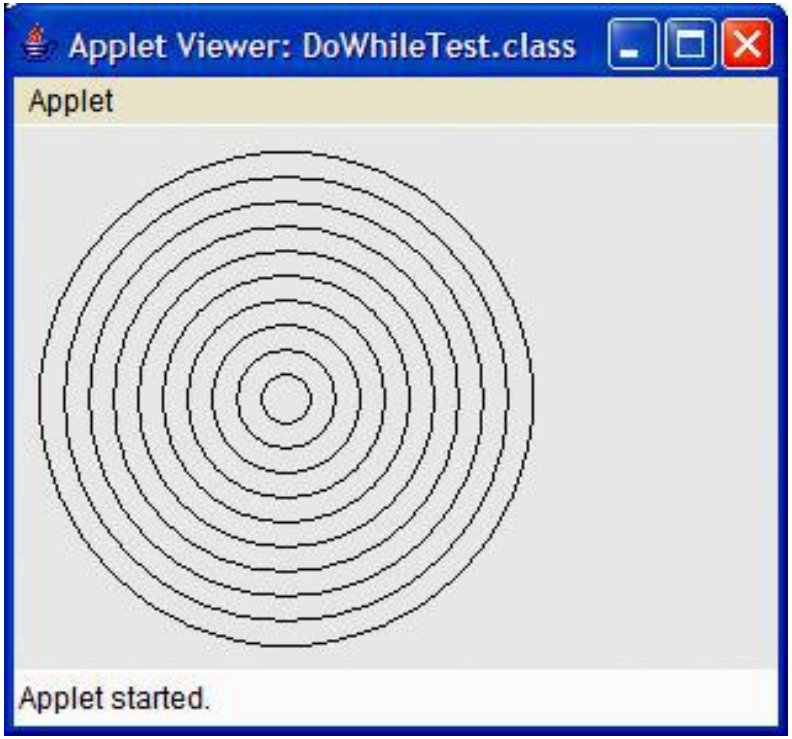


Figura 5.28 | Desenhando círculos concêntricos.

5.11 (Opcional) Estudo de caso de engenharia de software: Identificando estados e atividades dos objetos

- **Diagramas de máquina de estado:**
  - **Comumente chamado *diagrama de estado*.**
  - **Modela vários estados de um objeto.**
  - **Mostra sob quais circunstâncias o objeto muda de estado.**
  - **Focaliza o comportamento do sistema.**
  - **Representação UML:**
    - **Estado:**
      - **Retângulo arredondado.**
    - **Estado inicial:**
      - **Círculo sólido.**
    - **Transições:**
      - **Setas com pontas.**

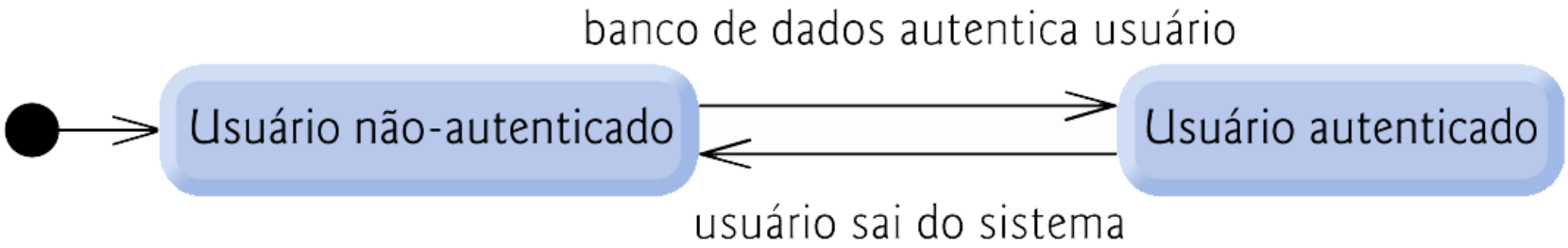


Figura 5.29 | Diagrama de classe para o objeto ATM.

Observação de engenharia de software 5.5

Em geral, os engenheiros de software não criam diagramas de estado que mostram cada estado possível e transição de estado para todos os atributos — há simplesmente muitos deles.

Os diagramas de estado em geral mostram apenas estados-chave e as principais transições de estado.

5.11 (Opcional) Estudo de caso de engenharia de software (*Continuação*)

- **Diagramas de atividade:**
  - **Focaliza o comportamento do sistema.**
  - **Modela o *fluxo de trabalho* de um objeto durante a execução do programa.**
  - **Modela as *ações* que o objeto realizará e em qual ordem.**
  - **Representação UML:**
    - **Estado da ação (retângulo com os lados esquerdos e direitos substituídos por arcos curvados para fora).**
    - **Ordem da ação (seta com ponta).**
    - **Estado inicial (círculo sólido).**
    - **Estado final (círculo sólido incluído em um círculo aberto).**

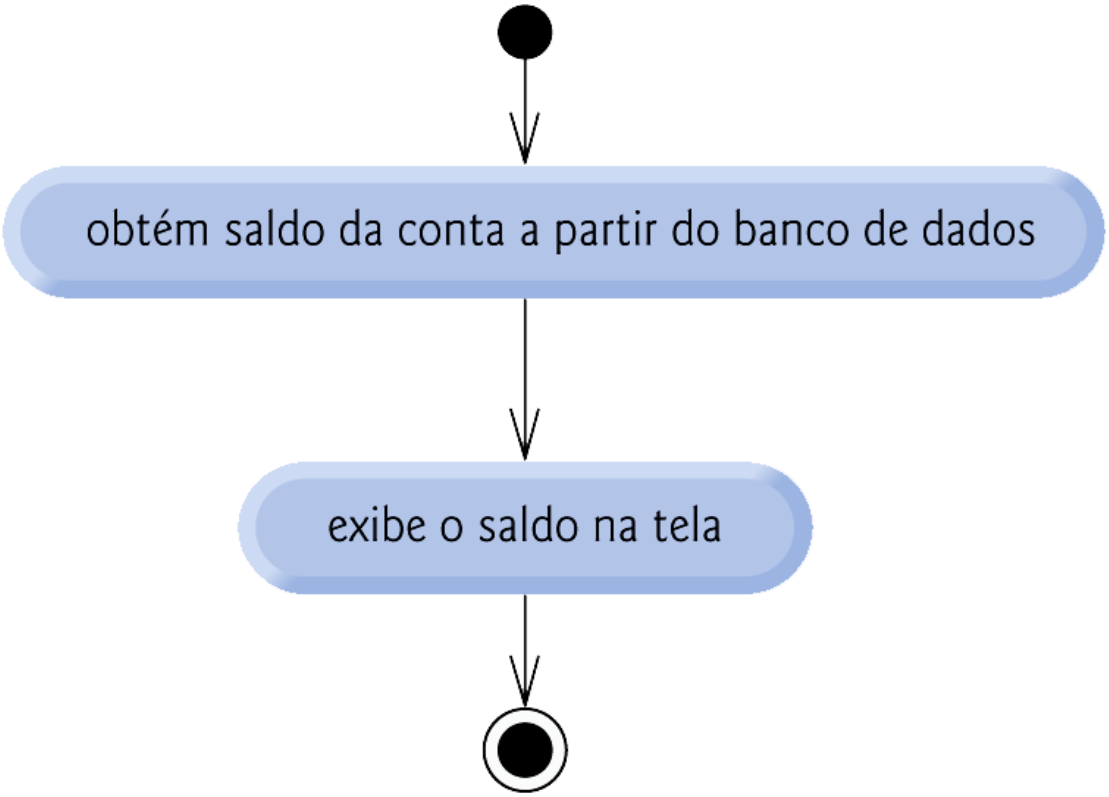


Figura 5.30 | Diagrama de atividades de um objeto `BalanceInquiry`.

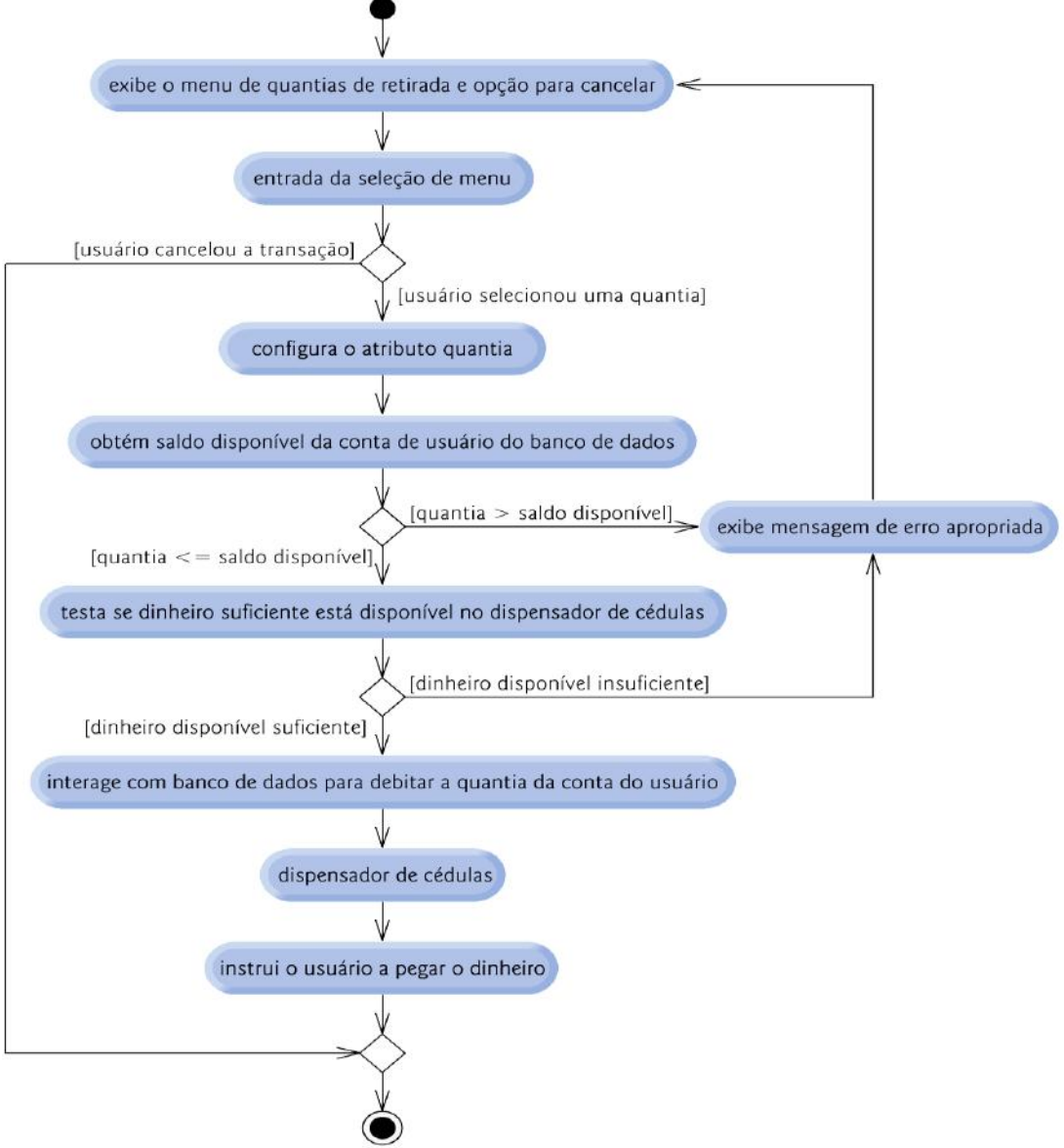


Figura 5.31 | Diagrama de atividades para uma transação de saque.

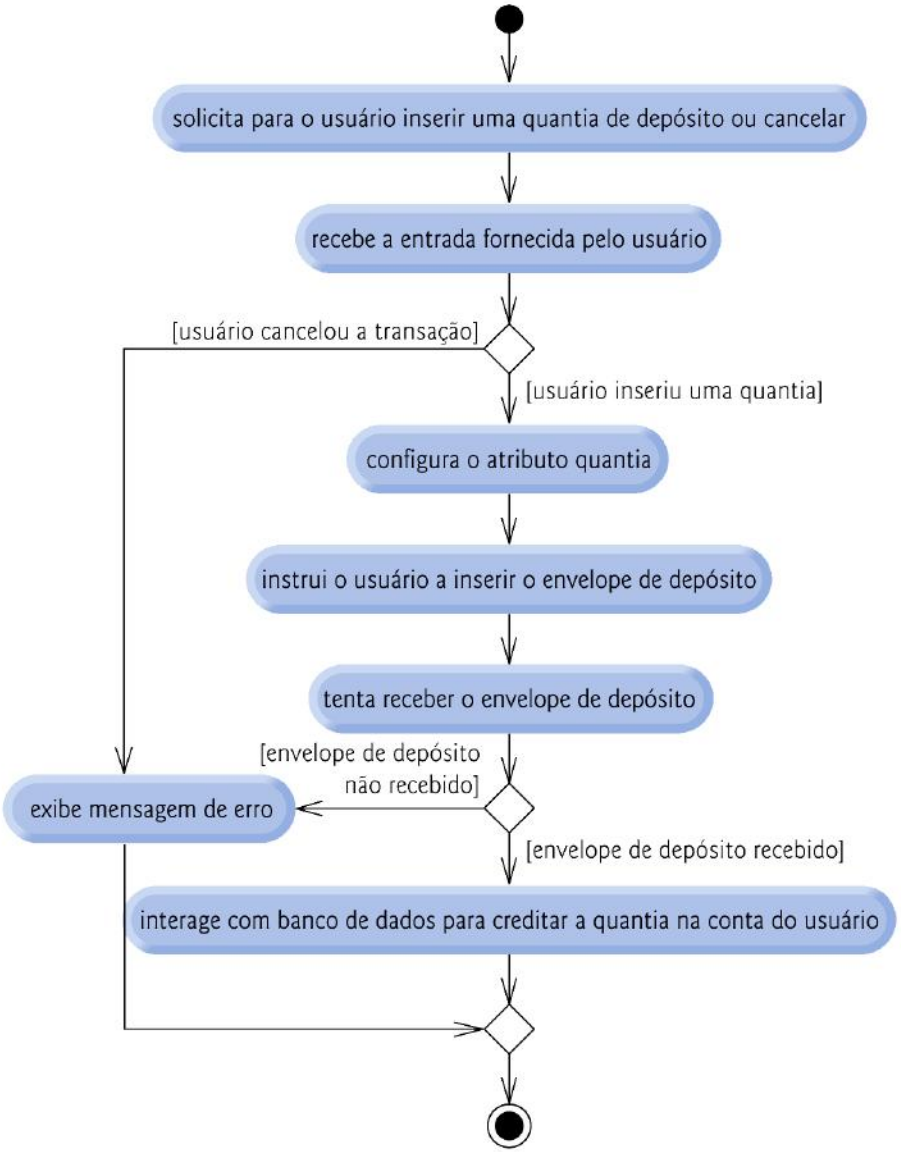


Figura 5.32 | Diagrama de atividades para uma transação de depósito.