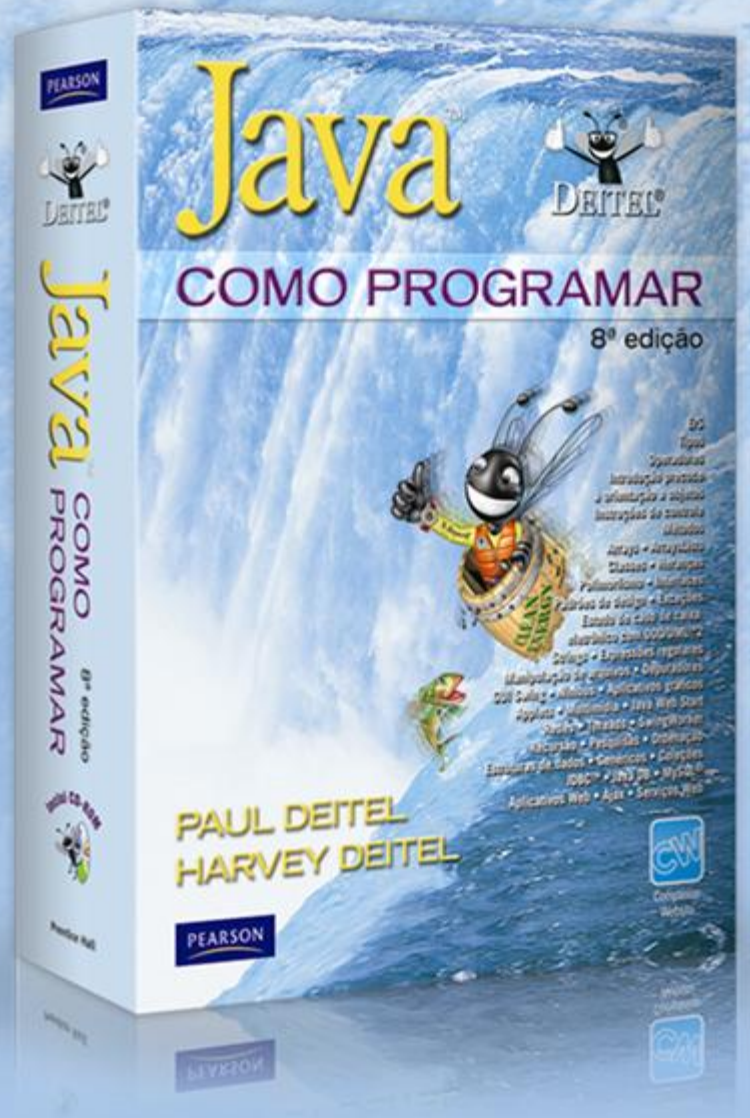
Capítulo 5 Instruções de controle: Parte 2

Java™ Como Programar, 8/E



OBJETIVOS

Neste capítulo, você aprenderá:

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

5.1 Introdução

- Instrução de repetição `for`
- Instrução de repetição `do...while`
- Instrução de seleção múltipla `switch`
- Instrução `break`
- Instrução `continue`
- Operadores lógicos
- Resumo de instruções de controle.

5.2 Princípios básicos de repetição controlada por contador

- Repetição controlada por contador requer:
  - uma **variável de controle** (ou contador de loop)
  - o **valor inicial** da variável de controle.
  - o **incremento** (ou **decremento**) pelo qual a variável de controle é modificada a cada passagem pelo loop (também conhecido como **cada iteração do loop**).
  - a **condição de continuação do loop** que determina se o loop deve continuar.


```
1  // Figura 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 de continuação do loop
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // incrementa a variável de controle por 1
14       } // fim do while
15
16       System.out.println(); // imprime uma nova linha
17    } // fim de main
18 } // fim da classe WhileCounter
```

Declara e inicializa a variável `counter` como 1

Condição de continuação do loop testa se é o valor final de `counter`

Inicializa a variável gradeCounter como 1; indica que a primeira nota será inserida

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

Figura 5.1 | Repetição controlada por contador com a instrução de repetição `while`.

- Na Figura 5.1, os elementos da repetição controlada por contador são definidos nas linhas 8, 10 e 13.
- A linha 8 declara a variável de controle (`counter`) como um `int`, reserva espaço para ele na memória e configura seu valor inicial como `1`.
- A condição de continuação do loop no `while` (linha 10) testa se o valor da variável de controle é menor que ou igual a `10` (o valor final para o qual a condição é `true`).
- A linha 13 incrementa a variável de controle por 1 para cada iteração do loop.

Java™ COMO PROGRAMAR 8ª edição

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 imprecisos e testes de terminação imprecisos.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 5.1

Utilize números inteiros para controlar loops de contagem.

Java™ COMO PROGRAMAR 8ª edição

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.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 5.1

"Manter a coisa simples" é um bom conselho para a maior parte do código que você escreverá.

5.3 Instrução de repetição for

- **Instrução de repetição** for
  - Especifica os detalhes da repetição controlada por contador em uma única linha de código.
  - A Figura 5.2 reimplementa o aplicativo da Figura 5.1 usando `for`.


```
1  // Figura 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(); // imprime uma nova linha
14    } // fim de main
15 } // fim da classe ForCounter
```

O cabeçalho da instrução `for` contém tudo que você precisa para uma repetição controlada por contador

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

Figura 5.2 | Repetição controlada por contador com a instrução de repetição `for`.

- Quando a instrução `for` começa a executar, a variável de controle é declarada e inicializada.
- Em seguida, o programa verifica a condição de continuação do loop, que está entre os dois ponto-e-vírgulas requeridos.
- Se inicialmente a condição for verdadeira, o corpo será executado.
- Depois de executar o corpo do loop, o programa incrementa a variável de controle na expressão de incremento, que aparece à direita do segundo ponto e vírgula.
- Então o teste de continuação do loop é realizado novamente para determinar se o programa deve continuar com a próxima iteração do loop.
- Um erro comum de lógica na repetição controlada por contador é um **erro off-by-one.**

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 5.2

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 por um.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 5.2

Usar o valor final na condição de uma instrução `while` *ou* `for` *e usar o operador relacional* `<=` *ajuda a evitar erros por um. 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 por um) ou* `counter < 11` *(o que é correto). Muitos programadores preferem a chamada contagem baseada em zero, em que para contar 10 vezes,* `counter` *seria inicializado como zero e o teste de continuação do loop seria* `counter < 10`.

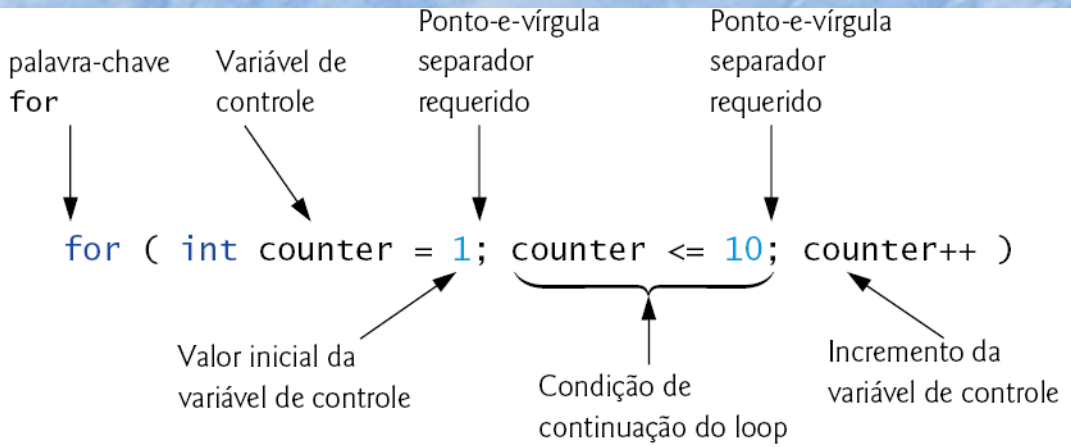


Figura 5.3 | Componentes de cabeçalho de instrução `for`.

- O formato geral da instrução `for` é

  `for` (*initialização*; *condiçãoDeContinuaçãoDoLoop*; *incremento*)
  instrução

  - a expressão *inicialização* nomeia a variável de controle do loop e fornece opcionalmente seu valor inicial.
  - *condiçãoDeContinuaçãoDoLoop* determina se o loop deve continuar executando.
  - *incremento* modifica o valor da variável de controle (possivelmente um incremento ou decremento), para que a condição de continuação do loop por fim se torne falsa.
- Os dois ponto e vírgulas no cabeçalho `for` são necessários.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 5.3

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

- Na maioria dos casos, a instrução `for` pode ser representada com uma instrução `while` equivalente como segue:

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

- Em geral, as instruções `for` são utilizadas para repetição controlada por contador e as instruções `while` são utilizadas para repetição controlada por sentinela.
- Se a expressão *inicialização* no cabeçalho `for` declara a variável de controle, a variável de controle pode ser usada só nessa instrução `for`.
- O **escopo** de uma variável define onde ele pode ser utilizado em um programa.
  - Uma variável local só pode ser utilizada no método que a declara e somente a partir do ponto de declaração até o fim do método.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 5.4

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

- Todas as três expressões em um cabeçalho `for` são opcionais.
  - Se a *condiçãoDeContinuaçãoDoLoop* for omitida, a condição é sempre verdadeira, criando assim um loop infinito.
  - Você poderia omitir a expressão *inicialização* se o programa inicializar a variável de controle antes do loop.
  - Você poderia omitir a expressão *incremento* se o programa o calcular com instruções no corpo do loop ou se nenhum incremento for necessário.
- A expressão incremento em uma instrução `for` atua como se ela fosse uma instrução independente no fim do corpo do `for`.

```
counter = counter + 1
counter += 1
++counter
counter++
```

são expressões de incremento equivalentes em uma instrução `for`.

Java™ COMO PROGRAMAR 8ª edição

Dica de desempenho 5.1

Há uma ligeira vantagem de desempenho em pré-incrementar, mas se você escolher pós-incrementar porque parece mais natural (como no cabeçalho de um `for`*), otimizar os compiladores normalmente irá gerar o bytecode Java que, de todo modo, utiliza a forma mais eficiente. Dessa forma, você deve utilizar o idioma com o qual você se sente mais à vontade nessas situações.*

Java™ COMO PROGRAMAR 8ª edição

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. Em geral, isso é um erro de lógica.*

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 5.3

Os loops infinitos ocorrem quando a condição de continuação do loop em uma instrução de repetição nunca 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 é capaz de ser inserido.*

- A inicialização, a condição de continuação de loop e p incremento podem conter expressões aritméticas.
- Por exemplo, assuma que `x = 2` e `y = 10`. Se `x` e `y` não forem modificados no corpo do loop, a instrução

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

- é equivalente à instrução

```
for (int j = 2; j <= 80; j += 5)
```

- O incremento de uma instrução `for` pode ser negativo, caso em que ele é, na verdade, um decremento e o loop conta para baixo.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 5.4

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.*

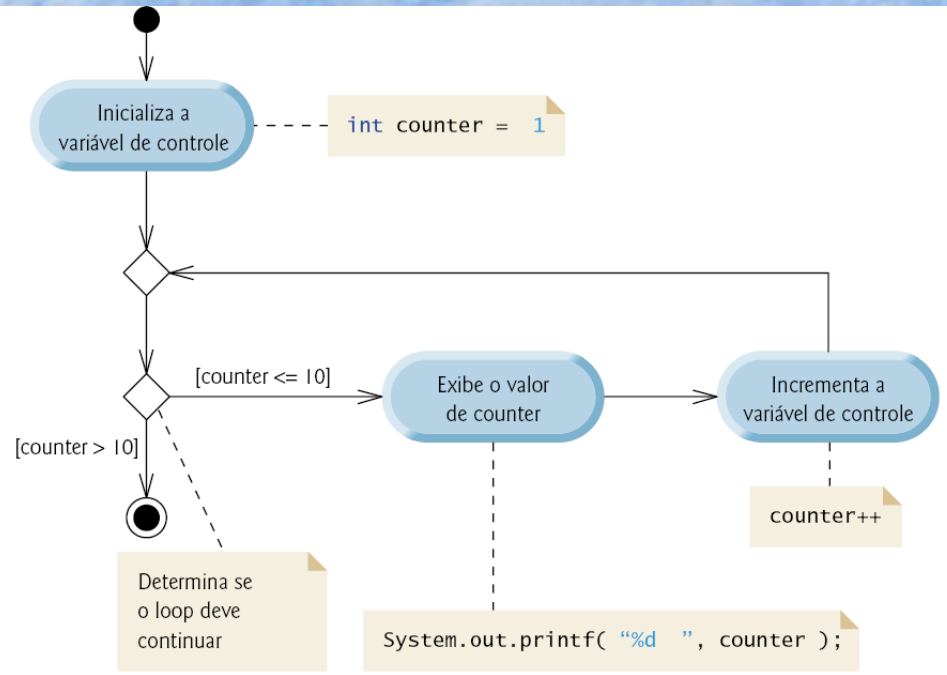


Figura 5.4 | Diagrama de atividades UML para a instrução `for` na Figura 5.2.

5.4 Exemplos com a estrutura for

- a) Varie a variável de controle de 1 a 100 em incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

- b) Varie a variável de controle de 100 a 1 em decrementos de 1.

```
for ( int i = 100; i >= 1; i-- )
```

- c) Varie a variável de controle de 7 a 77 em incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

- d) Varie a variável de controle de 20 a 2 em decrementos de 2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

- e) Varie a variável de controle pelos valores 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

- f) Varie a variável de controle pelos valores 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 5.6

Utilizar um operador relacional incorreto 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.*


```
1  // Figura 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
```

Observe que o incremento neste loop é 2, uma vez que queremos totalizar somente os inteiros pares

```
Sum is 110
```

Figura 5.5 | Somando inteiros com a instrução `for`.

- As expressões *inicialização* e *incremento* podem ser listas separadas por vírgulas de expressões que permitem-lhe utilizar múltiplas expressões de inicialização ou múltiplas expressões de incremento.
- Por exemplo, embora não seja aconselhável, o corpo da instrução `for` nas linhas 11–12 da Figura 5.5 poderia ser mesclado na parte de incremento do cabeçalho `for` utilizando uma vírgula, como a seguir:

```
for ( int number = 2;
   number <= 20;
   total += number, number += 2 )
   ; // estrutura vazia
```

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 5.2

Para maior legibilidade, limite o tamanho de cabeçalhos da instrução de controle a uma única linha se possível.

- Aplicativo de juros compostos
- *Uma pessoa investe $1.000 em uma conta-poupança que rende juros de 5% ao ano. Assumindo que todo o juro é deixado em depósito, calcule e imprima a quantidade de dinheiro na conta no fim de cada ano por 10 anos. Utilize a seguinte fórmula para determinar as quantidades:*

  $a = p\ (1 + r)^n$

  onde

  p é a quantidade original investida (isto é, o principal)
  r é a taxa de juros anual (por exemplo, utilize 0.05 para 5%)
  n é o número de anos
  a é a quantidade em depósito ao fim do n-ésimo ano.

- A solução para esse problema (Figura 5.6) envolve um loop que realiza o cálculo indicado para cada um dos 10 anos que o dinheiro permanece em depósito.
- O Java trata constantes de ponto flutuante como `1000.0` e `0.05` como tipo `double`.
- O Java trata constantes inteiras como `7` e `-22` como tipo `int`.

```
1  // Figura 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; // quantidade 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
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
```

O Java trata literais de ponto flutuante como valores double

Usa o método static Math.pow para ajudar a calcular a quantia em depósito

Figura 5.6 | Cálculos de juros compostos com `for`. (Parte 1 de 2.)


```
21            // exibe o ano e a quantidade
22            System.out.printf( "%4d%,20.2f\n", year, amount );
23         } // for final
24      } // fim de main
25   } // fim da classe Interest
```

Vírgula no especificador de formato indica que números grandes devem ser exibidos com separadores de milhar

```
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
```

Figura 5.6 | Cálculos de juros compostos com `for`. (Parte 2 de 2.)

- No especificador de formato `%20s`, o inteiro `20` entre o `%` e o caráter de conversão `s` indica que a saída do valor deve ser exibida com uma **largura de campo** de 20 — isto é, `printf` exibe o valor com pelo menos 20 posições de caractere.
- Se o valor a ser enviado para a saída for menor do que a largura de 20 posições de caractere, o valor é **alinhado à direita** no campo por padrão.
- Se o valor de `year` a ser enviado para a saída tiver mais caracteres que a largura do campo, a largura do campo será estendida à direita para acomodar todo o valor.
- Para indicar que os valores devem ser enviados para a saída **alinhados à esquerda**, simplesmente preceda a largura de campo com o **flag de formatação sinal de subtração (–)** (e.g., `%-20s`).

- Classes fornecem métodos que executam tarefas comuns sobre objetos.
- A maioria dos métodos deve ser chamada sobre um objeto específico.
- Muitas classes também fornecem métodos que realizam tarefas comuns e não exigem objetos. Estes são chamados métodos `static`.
- o Java não inclui um operador de exponenciação — o método `static` `pow` da classe `Math` pode ser usado para elevar um valor a uma potência.
- Você pode chamar um método `static` especificando o nome da classe seguido por um ponto (`.`) e o nome de método, assim
  - *NomeDaClasse*`.`*nomeDoMétodo*`(` *argumentos* `)`
- `Math.pow(`*x*`,` *y*`)` calcula o valor de *x* elevado à *y*-ésima potência. O método recebe dois argumentos `double` e retorna um valor `double`.

Java™ COMO PROGRAMAR 8ª edição

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.*]

- No especificador de formato `%,20.2f`, o **flag de formatação vírgula (,)** indica que um valor de ponto flutuante deve ser enviado para a saída com um **separador de agrupamento.**
- O separador real utilizado é específico à localidade do usuário (isto é, país).
- Nos Estados Unidos, o número será enviado para saída utilizando vírgulas para separar cada três dígitos e um ponto de fração decimal para separar a parte fracionária do número, como em 1,234.45.
- O número `20` na especificação de formato indica que o valor deve ser enviado para a saída alinhado à direita em uma largura de campo de 20 caracteres.
- O `.2` especifica a precisão do número formatado — nesse caso, o número é arredondado para o centésimo mais próximo e enviado para saída com dois dígitos à direita do ponto de fração decimal.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 5.5

Não utilizar variáveis de tipo `double` (*ou* `float`) *para realizar cálculos monetários precisos. A imprecisão dos números de ponto flutuante pode causar erros. Nos exercícios, você aprenderá como usar números inteiros para realizar cálculos monetários precisos. O Java também fornece a classe* `java.math.BigDecimal` *para realizar cálculos monetários precisos. Para obter mais informações, consulte:*

`java.sun.com/javase/6/docs/api/java/math/BigDecimal.html`.

5.5 Instrução de repetição do...while

- A **instrução de repetição `do...while`** é semelhante à instrução `while`.
- Na instrução `while`, o programa testa a condição de continuação do loop no início do loop, antes de executar o corpo do loop; se a condição for falsa, o corpo nunca será executado.
- A instrução `do…while` testa a condição de continuação do loop *depois de executar o corpo do loop*; portanto, o corpo sempre executa pelo menos uma vez.
- Quando uma instrução `do…while` termina, a execução continua com a próxima instrução na sequência.

```
1  // Figura 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 o contador
9
10       do
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter;
14       } while ( counter <= 10 ); // fim da instrução do…while
15
16       System.out.println(); // gera saída de um caractere nova linha
17    } // fim de main
18 } // fim da classe DoWhileTest
```

Condição testada no final do loop; portanto, o loop executa pelo menos uma vez

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

Figura 5.7 | Instrução de repetição do...while.

- A Figura 5.8 contém o diagrama de atividades UML para a instrução `do...while`.
- O diagrama torna claro que a condição de continuação do loop não é avaliada enquanto o loop não executar o estado de ação pelo menos uma vez.

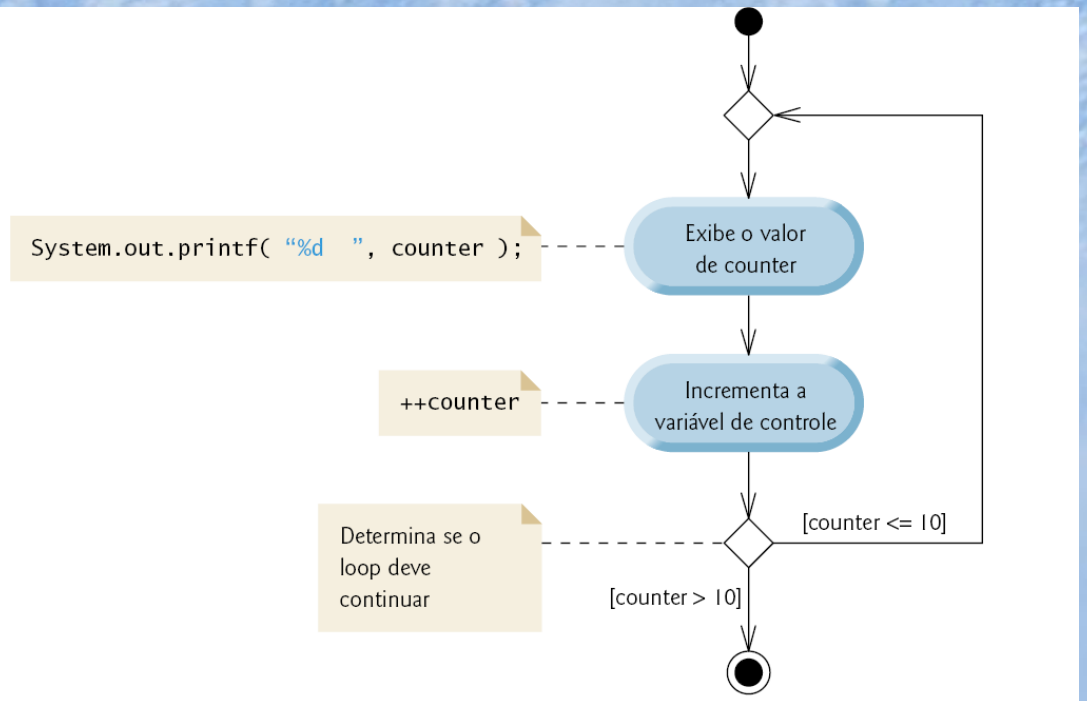


Figura 5.8 | Diagrama de atividades UML da instrução de repetição do...while.

- As chaves não são obrigatórias na instrução de repetição `do...while` se houver apenas uma instrução no corpo.
- A maioria dos programadores inclui as chaves, para evitar confusão entre as instruções `while` e `do...while`.
- Portanto, a instrução `do...while` com uma instrução de corpo é normalmente escrita assim:

```
do
{
   instrução
} while ( condição );
```

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 5.3

Sempre inclua chaves em uma instrução `do...while`*, mesmo se não forem necessárias. Isso ajuda a eliminar ambiguidade entre a instrução* `while` *e uma instrução* `do...while` *que contém apenas uma instrução.*

5.6 Instrução de seleção múltipla `switch`

- A **instrução de seleção múltipla** `switch` realiza ações diferentes com base nos possíveis valores de uma **expressão inteira constante** do tipo `byte`, `short`, `int` ou `char`.


```
1  // Figura 5.9: GradeBook.java
2  // A classe GradeBook utiliza a instrução switch para contar as letras das notas.
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     // variáveis de instância int são inicializadas em 0 por padrão
9     private int total; // soma das notas
10    private int gradeCounter; // número de notas inseridas
11    private int aCount; // contagem de notas A
12    private int bCount; // contagem de notas B
13    private int cCount; // contagem de notas C
14    private int dCount; // contagem de notas D
15    private int fCount; // contagem de notas F
16
17    // construtor inicializa courseName;
18    public GradeBook( String name )
19    {
20       courseName = name; // inicializa courseName
21    } // fim do construtor
22
```

Não é ncessário inicializar esses totais e contadores como 0, embora muitos programadores considerem isso uma boa prática de programação

Figura 5.9 | A classe `GradeBook` utiliza a instrução `switch` para contar as letras das notas escolares. (Parte 1 de 6.)


```
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
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
```

Figura 5.9 | A classe `GradeBook` utiliza a instrução `switch` para contar as letras das notas escolares. (Parte 2 de 6.)


```
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
56         // faz loop até usuário inserir o indicador de fim do arquivo
57         while (input.hasNext() )
58         {
59            grade = input.nextInt(); // lê a nota
60            total += grade; // adiciona grade a total
61            ++gradeCounter; // incrementa o número de notas
62
63            // chama método para incrementar o contador adequado
64            incrementLetterGradeCounter( grade );
65         } // fim do while
66      } // fim do método inputGrades
```

O loop continua até o indicador de final de arquivo ser encontrado

Figura 5.9 | A classe `GradeBook` utiliza a instrução `switch` para contar as letras das notas escolares. (Parte 3 de 6.)


```
67
68      // adiciona 1 ao contador adequado da nota especificada
69      private void incrementLetterGradeCounter( int grade )
70      {
71         // determina a nota que foi inserida
72         switch ( grade / 10 )
73         {
74            case 9:  // a nota estava entre 90
75            case 10: // e 100, inclusivo
76               ++aCount; // incrementa aCount
77               break; // necessário para sair de switch
78
79            case 8: // nota estava entre 80 e 89
80               ++bCount; // incrementa bCount
81               break; // sai do switch
82
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 e 69
88               ++dCount; // incrementa dCount
89               break; // sai do switch
90
```

grade/10 é a expressão de controle; o valor inteiro resultante é comparado com o valor de cada rótulo case

Figura 5.9 | A classe GradeBook utiliza a instrução switch para contar as letras das notas escolares. (Parte 4 de 6.)

```
91             default: // a nota era menor que 60
92                ++fCount; // incrementa fCount
93                break; // opcional; sairá do switch de qualquer jeito
94             } // fim do switch
95          } // fim do método incrementLetterGradeCounter
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
108               // gera a saída de resumo de 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 );
```

O caso `default` executa para notas inferiores a 60

Testa para a possibilidade de divisão por zero

Figura 5.9 | A classe `GradeBook` utiliza a instrução `switch` para contar as letras das notas escolares. (Parte 5 de 6.)


```
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 // notas não foram inseridas, portanto imprime mensagem apropriada
121             System.out.println( "No grades were entered" );
122       } // fim do método displayGradeReport
123    } // fim da classe GradeBook
```

Figura 5.9 | A classe `GradeBook` utiliza a instrução `switch` para contar as letras das notas escolares. (Parte 6 de 6.)


```
1  // Figura 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 myGradeBook da classe GradeBook e
9        // passa o nome do 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 notas fornecidas pelo usuário
15       myGradeBook.displayGradeReport(); // exibe relatório baseado em notas
16    } // fim de main
17 } // fim da classe GradeBookTest
```

Chama os métodos **public** de **GradeBook** para inserir e resumir as notas e, então, exibe o relatório de notas

Figura 5.10 | **GradeBookTest** cria um objeto **GradeBook** e invoca seus métodos. (Parte 1 de 3.)


```
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
```

Figura 5.10 | `GradeBookTest` cria um objeto `GradeBook` e invoca seus métodos. (Parte 2 de 3.)


```
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
```

Figura 5.10 | `GradeBookTest` cria um objeto `GradeBook` e invoca seus métodos. (Parte 3 de 3.)

- O **indicador de fim do arquivo** é uma combinação de pressionamentos de tecla dependente de sistema que o usuário insere para indicar que não há dados a serem inseridos.
- Nos sistemas UNIX/Linux/Mac OS X, o fim do arquivo é inserido digitando a seqüência
  - *<Ctrl> d*
- em uma linha isolada. Essa notação significa pressionar simultaneamente a tecla *Ctrl* e a tecla *d*.
- Em sistemas Windows, o fim do arquivo pode ser inserido digitando
  - *<Ctrl> z*
- Em alguns sistemas, você deve pressionar *Enter* depois de digitar a sequência de teclas de fim do arquivo.
- O Windows normalmente exibe os caracteres ^Z na tela quando o indicador de fim do arquivo é digitado.

Java™ COMO PROGRAMAR 8ª edição

Dica de portabilidade 5.1

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

- O método `Scanner` `hasNext` determina se há mais dados a inserir. Esse método retorna o valor `booleantrue` se houver mais dados; do contrário, ele retorna `false`.
- Enquanto o indicador de fim do arquivo não tiver sido digitado, o método `hasNext` retornará `true`.

- A instrução `switch` consiste em um bloco que contém uma sequência de rótulos `case` e um **`case default`.**
- O programa avalia a **expressão de controle** entre os parênteses que se seguem à palavra-chave `switch`.
- O programa compara o valor da expressão controladora (que deve ser avaliada como um valor integral do tipo `byte`, `char`, `short` ou `int`) com cada rótulo `case`.
- Se ocorrer uma correspondência, o programa executará as instruções para esse `case`.
- A **instrução `break`** faz com que o controle do programa prossiga para a primeira instrução depois do `switch`.

- `switch` não fornece um mecanismo para testar intervalos de valores — cada valor que deve ser testado deve ser listado em um rótulo `case` separado.
- Observe que cada `case` pode ter múltiplas instruções.
- `switch` difere de outras instruções de controle porque não exige que as múltiplas instruções em um `case` estejam entre chaves.
- Sem um `break`, as instruções para um caso correspondente e casos subsequentes são executadas até que uma instrução `break` ou o fim do `switch` seja encontrado. Isso é chamado de "falling through".
- Se não ocorrer nenhuma correspondência entre o valor da expressão controladora e um rótulo `case`, o caso `default` opcional é executado.
- Se não ocorrer nenhuma correspondência e não houver um caso `default`, o controle de programa simplesmente continua com a primeira instrução depois do `switch`.

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 5.7

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

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 5.2

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

- A Figura 5.11 mostra o diagrama de atividades UML para a instrução `switch` geral.
- A maioria das instruções `switch` usa um `break` em cada `case` para terminar a instrução `switch` depois de processar o `case`.
- A instrução `break` não é necessária para o último `case` do `switch` (ou o `case` default opcional, quando ele aparece por último), porque a execução continua com a próxima instrução depois do `switch`.

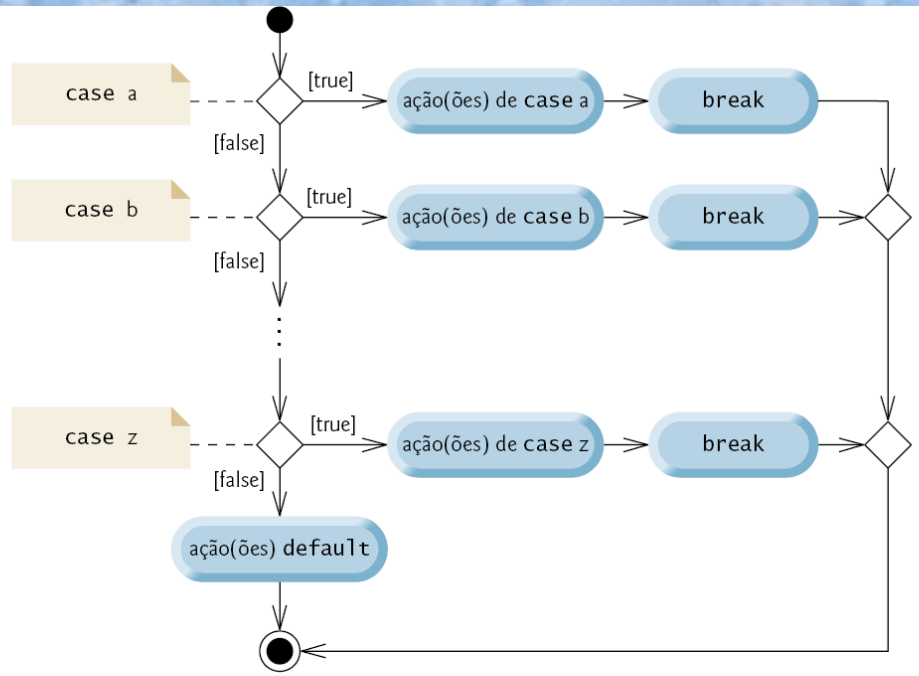


Figura 5.11 | Diagrama de atividades UML de instrução de seleção múltipla `switch` com instruções `break`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 5.3

Forneça um case `default` *nas instruções* `switch`*. Incluir um caso* `default` *faz com que você se concentre na necessidade de processar condições excepcionais.*

Java™ COMO PROGRAMAR 8ª edição

Boa prática de programação 5.4

Embora cada `case` *e o caso* `default` *em uma* `switch` *possam ocorrer em qualquer ordem, coloque o caso* `default` *por último. Quando o caso* `default` *é listado por último, o* `break` *para esse caso não é necessário.*

- Ao utilizar a instrução `switch`, lembre-se de que cada `case` deve conter uma expressão integral constante.
- Uma constante integral é simplesmente um valor inteiro.
- Além disso, você pode utilizar **constantes de caractere** — caracteres específicos entre aspas simples, como `'A'`, `'7'` or `'$'` — que representam os valores inteiros dos caracteres.
- A expressão em cada `case` também pode ser uma **variável constante** — uma variável que contém um valor que não muda no programa inteiro. Essa variável é declarada com a palavra-chave `final`.
- O Java tem um recurso chamado enumerações. Constantes de enumeração também podem ser utilizadas em rótulos `case`.

5.7 Instruções `break` e `continue`

- A instrução `break` quando executada em um `while`, `for`, `do...while` ou `switch`, ocasiona a saída imediata dessa instrução.
- A execução continua com a primeira instrução depois da instrução de controle.
- Usos comuns da instrução `break` são escapar antecipadamente de um loop ou pular o restante de uma instrução `switch`.


```
1  // Figura 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 usada depois que o loop termina
8
9        for ( count = 1; count <= 10; count++ ) // faz o loop 10 vezes
10       {
11          if ( count == 5 ) // se contagem for 5,
12             break;         // termina o loop
13
14          System.out.printf( "%d ", count );
15       } // for final
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // fim de main
19 } // fim da classe BreakTest
```

Termina o loop imediatamente e o controle do programa continua na linha 17

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

Figura 5.12 | Instrução `break` saindo de uma instrução `for`.

- A instrução `continue`, quando executada em um `while`, `for` ou `do`…`while`, pula as instruções restantes no corpo do loop e prossegue com a próxima iteração do loop.
- Nas instruções `while` e `do`…`while`, o programa avalia o teste de continuação do loop imediatamente depois que a instrução `continue` é executada.
- Em uma instrução `for`, a expressão incremento é executada, então o programa avalia o teste de continuação do loop.


```
1  // Figura 5.13: ContinueTest.java
2  // continua instrução que termina 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++ ) // faz o loop 10 vezes
8        {
9           if ( count == 5 ) // se contagem for 5,
10             continue; // pula o código restante no loop
11
12          System.out.printf( "%d ", count );
13       } // for final
14
15       System.out.println( "\nUsed continue to skip printing 5" );
16    } // fim de main
17 } // fim da classe ContinueTest
```

Termina a iteração atual do loop e prossegue com o incremento

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

Figura 5.13 | Instrução `continue` terminando uma iteração de uma instrução `for`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 5.4

Alguns programadores acham que `break` *e* `continue` *violam a programação estruturada. Como os mesmos efeitos são alcançáveis com as técnicas de programação estruturada, esses programadores não utilizam* `break` *ou* `continue`.

Java™ COMO PROGRAMAR 8ª edição

Observação de engenharia de software 5.5

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

- Os **operadores lógicos** do Java permitem formar condições mais complexas combinando condições simples.
- Os operadores lógicos são
  - && (E condicional)
  - || (OU condicional)
  - & (E lógico booleano)
  - | (OU lógico booleano inclusivo)
  - ^ (Ou lógico booleano exclusivo)
  - ! (NOT lógico).
- [*Nota:* Os operadores &, | e ^ também são operadores de bits quando eles são aplicados a operandos de números inteiros.

- O operador && (**E condicional)** certifica-se de que duas condições são *ambas verdadeiras* antes de escolher certo caminho de execução.
- A tabela na Figura 5.14 resume o operador **&&**. A tabela mostra todas as quatro possíveis combinações de valores `false` e `true` para *expressão1* e *expressão2*.
- Essas tabelas são chamadas de **tabelas-verdade**. O Java avalia todas as expressões `false` ou `true` que incluem operadores relacionais, operadores de igualdade ou operadores lógicos.

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).

- O operador || (**OU condicional**) certifica-se de que *uma ou ambas* as condições são verdadeiras antes de escolher certo caminho de execução.
- A Figura 5.15 é uma tabela-verdade para o operador OU condicional (||).
- O operador && tem uma precedência mais alta do que operador ||.
- Ambos os operadores associam-se da esquerda para direta.

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).

- As partes de uma expressão contendo os operadores **&&** ou **||** só são avaliadas até que se saiba se a condição é verdadeira ou falsa.
- Esse recurso das expressões E condicional e OU condicional chama-se **avaliação em curto-circuito.**

Java™ COMO PROGRAMAR 8ª edição

Erro comum de programação 5.8

Em expressões que usam 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. Nesse caso, a condição dependente deve ser colocada depois da outra condição, para não ocorrer um erro. Por exemplo, na expressão `( i != 0 ) && ( 10 / i == 2 )`*, a segunda condição deve aparecer depois da primeira para não ocorrer um erro de divisão por zero.*

- Os operadores **E lógico booleano** (**&**) e **OU lógico booleano inclusivo** (**|**) são idênticos aos operadores **&&** e **||**, exceto que os operadores **&** e **|** *sempre avaliam ambos seus operadores* (isto é, eles não realizam avaliação de curto-circuito).
- Isso é útil se o operando à direita do operador lógico booleano AND ou do operador lógico booleano OU inclusivo, tiverem um **efeito colateral** requerido — uma modificação no valor de uma variável.

Java™ COMO PROGRAMAR 8ª edição

Dica de prevenção de erro 5.6

Por clareza, evite expressões com efeitos colaterais nas condições. Efeitos colaterais talvez pareçam inteligentes, mas dificultam o entendimento do código e levam a erros de lógica sutis.

- Uma condição simples que contém o operador **OU lógico booleano exclusivo** (^) é `true` *se e somente se* um dos seus operados for `true` e o outro for `false`.
- Se ambos forem `true` ou ambos forem `false`, a condição inteira é `false`.
- A Figura 5.16 é uma tabela-verdade para o operador OR lógico booleano exclusivo (^).
- É garantido que esse operador avaliará seus dois operandos.

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

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

- O ! (**NÃO lógico**, também chamado **negação lógica** ou **complemento lógico**) o operador "inverte" o significado de uma condição.
- O operador lógico de negação tem apenas uma única condição como um operando.
- O operador de negação lógica é colocado antes de uma condição escolher um caminho de execução se a condição original (sem o operador de negação lógica) for `false`.
- Na maioria dos casos, você pode evitar a utilização da negação lógica expressando a condição diferentemente com um operador relacional ou de igualdade apropriado.
- A Figura 5.17 é uma tabela-verdade para o operador lógico de negação.

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

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

- A Figura 5.18 produz as tabelas de verdade discutidas nesta seção.
- Observe que utilizamos o **especificador de formato** `%b` para exibir a palavra "true" ou a palavra "false" com base em um valor `boolean` da expressão.

```
1  // Figura 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
```

O valor de cada condição como está é exibido usando o especificador de formato %b

Figura 5.18 | Operadores lógicos. (Parte 1 de 4.)

```
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
29          // cria a tabela-verdade para o operador | (OU inclusivo lógico booleano)
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 lógico booleano exclusivo)
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
```

Figura 5.18 | Operadores lógicos. (Parte 2 de 4.)

```
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
```

```
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
```

Figura 5.18 | Operadores lógicos. (Parte 3 de 4.)

```
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
```

Figura 5.18 | Operadores lógicos. (Parte 4 de 4.)

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 lógico booleano inclusivo
&&	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

- A Figura 5.20 utiliza diagramas de atividade UML para resumir instruções de controle do Java.
- O Java inclui apenas instruções de controle de entrada única/saída única — há somente uma maneira de entrar e somente uma maneira de sair de cada instrução de controle.
- É simples conectar instruções de controle em sequência para formar programas estruturados. O estado final de uma instrução de controle é conectado ao estado inicial da próxima — isto é, as instruções de controle são colocadas uma depois da outra em um programa em sequência. Chamamos isso de empilhamento de instruções de controle.
- As regras para formar programas estruturados também permitem que instruções de controle sejam aninhadas.

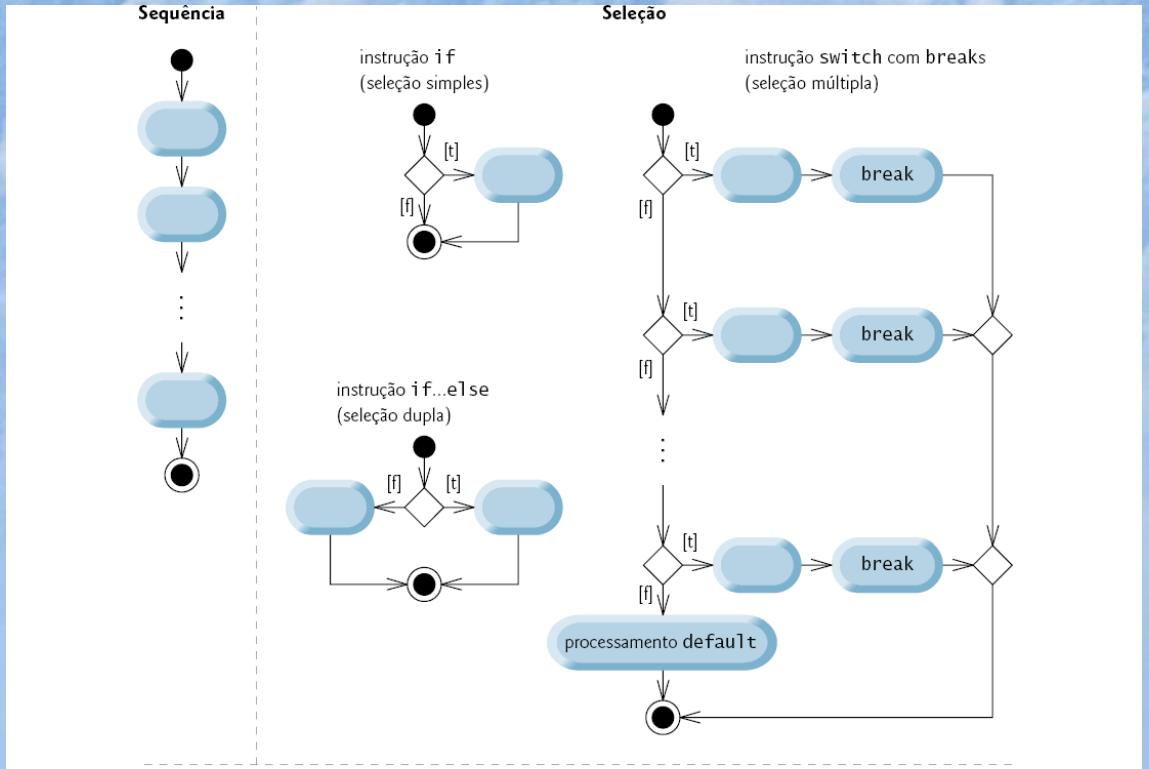


Figura 5.20 | As instruções de sequência de entrada única/saída única, seleção e repetição do Java. (Parte 1 de 2.)

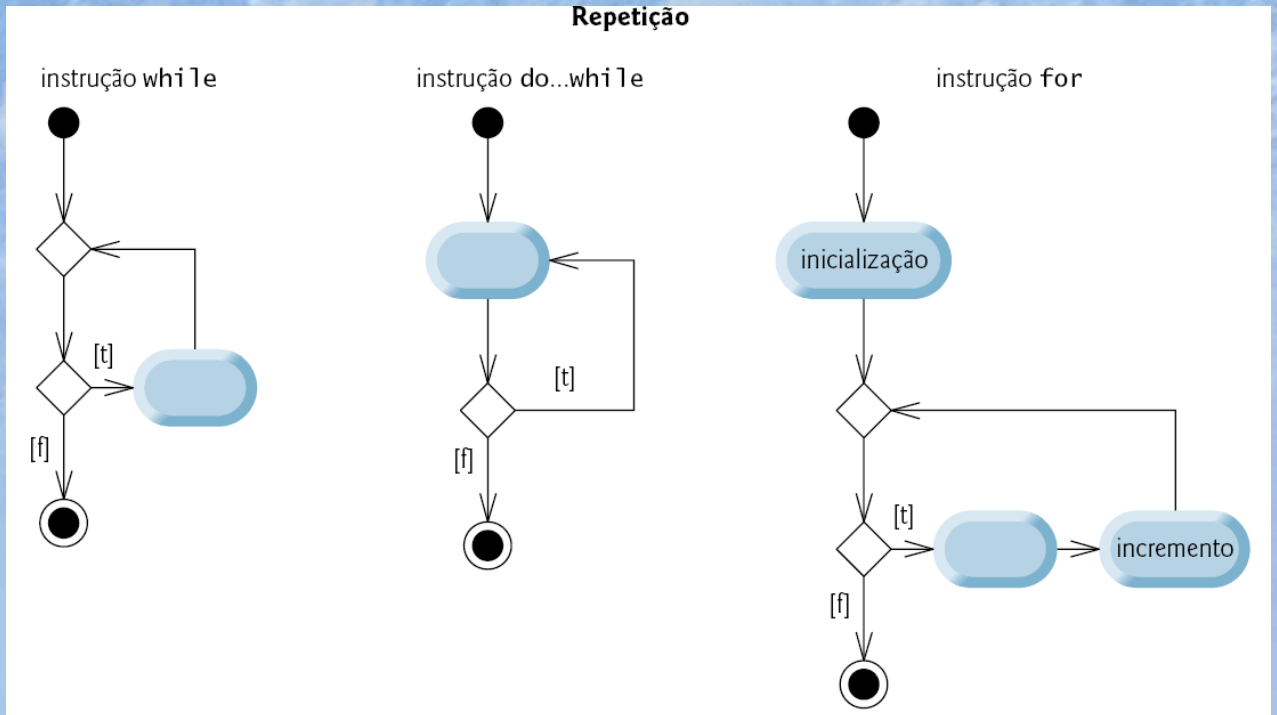


Figura 5.20 | As instruções de sequência de entrada única/saída única, seleção e repetição do Java. (Parte 2 de 2.)

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

Figura 5.21 | As regras para formar programas estruturados.

Figura 5.22 | Diagrama de atividades mais simples.

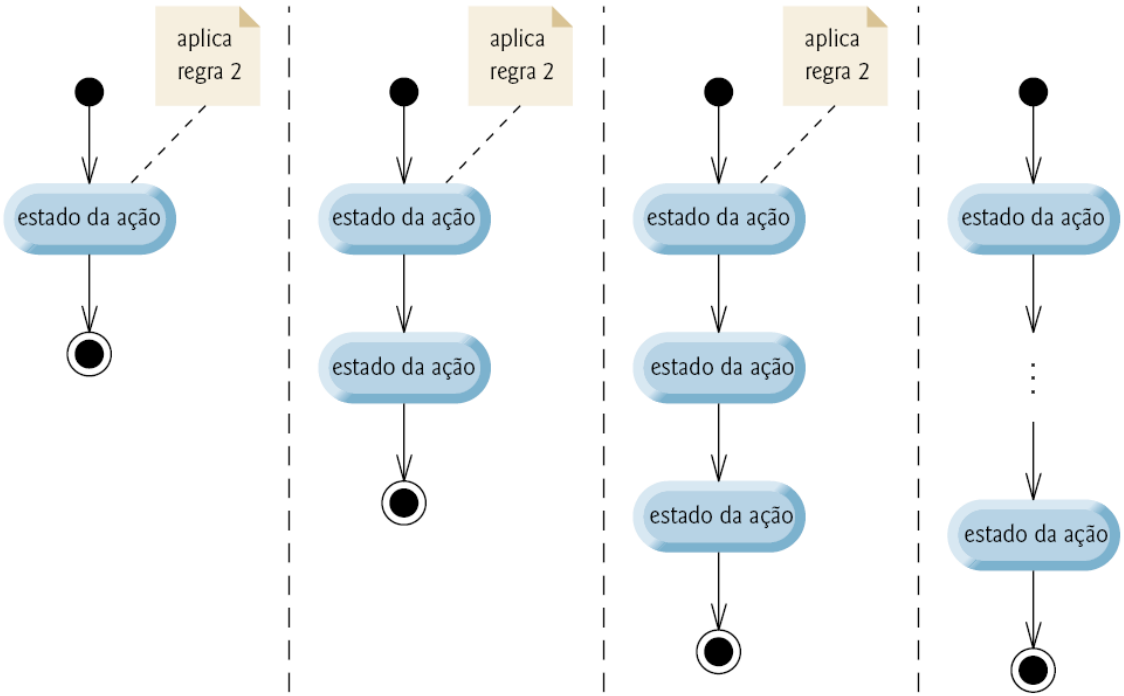


Figura 5.23 | Aplicando a regra de empilhamento repetidamente (regra 2) da Figura 5.21 ao diagrama de atividades mais simples.

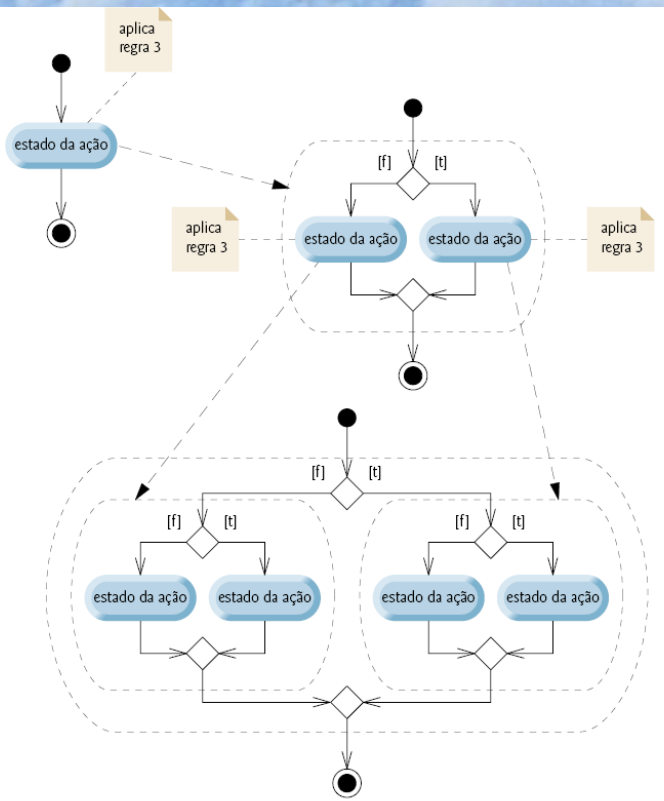


Figura 5.24 | Aplicando a regra de aninhamento repetidamente (regra 3) da Figura 5.21 ao diagrama de atividades mais simples.

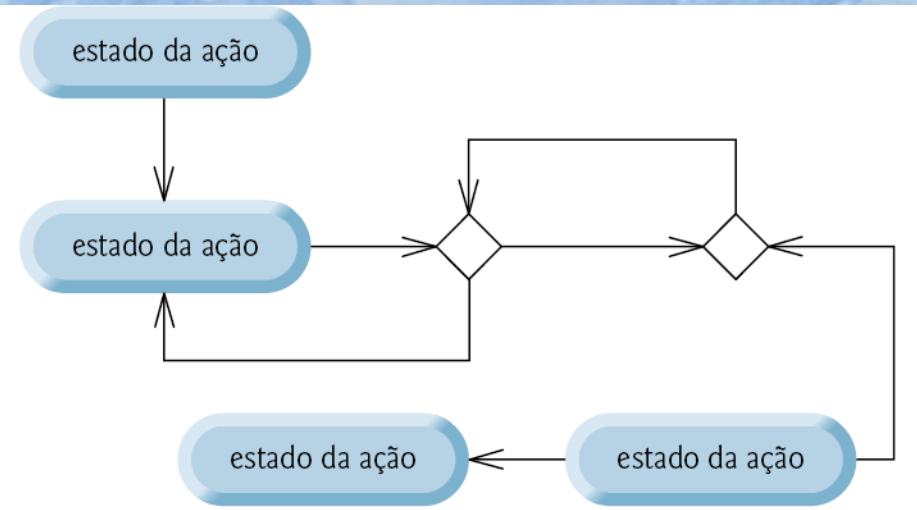


Figura 5.25 | Diagrama de atividades "não estruturado".

- Programação estruturada promove a simplicidade.
- Bohm e Jacopini: Apenas três formas de controle são necessárias para implementar um algoritmo:
  - Sequência
  - Seleção
  - Repetição
- A estrutura de sequência é trivial. Liste simplesmente as instruções para executar na ordem em que elas devem executar.

- A seleção é implementada de uma destas três maneiras:
  - Instrução `if` (seleção simples)
  - Instrução `if...else` (seleção dupla)
  - Instrução `switch` (seleção múltipla)
- A simples instrução `if` é suficiente para fornecer qualquer forma de seleção — tudo que pode ser feito com a instrução `if...else` e a instrução `switch` pode ser implementado combinando-se instruções `if`.

- A repetição é implementada de uma destas três maneiras:
  - Instrução `while`
  - Instrução `do...while`
  - Instrução `for`
- A instrução `while` é suficiente para fornecer qualquer forma de repetição. Tudo o que pode ser feito com `do...while` e `for` pode ser feito com a instrução `while`.

- A combinação desses resultados ilustra que qualquer forma de controle que possa ser necessária um dia em um programa Java pode ser expressa em termos de
  - sequência
  - Instrução `if` (seleção)
  - Instrução `while` (repetição)

  e estas podem ser combinadas apenas de duas maneiras — empilhamento e aninhamento.

5.10 (Opcional) Estudo de caso de GUI e imagens gráficas: desenhando retângulos e ovais

- Métodos **drawRect** e drawOval de `Graphics`.
- O método `drawRect` requer quatro argumentos. As duas primeiras representam as coordenadas *x* e *y* do canto superior esquerdo do retângulo; as duas seguintes representam a largura e altura do retângulo.
- Para desenhar uma oval, o método `drawOval` cria um retângulo imaginário chamado **retângulo delimitador** e posiciona dentro dele uma oval que toca os pontos centrais dos quatro lados.
- O método `drawOval` requer os mesmos quatro argumentos que o método `drawRect`. Os argumentos especificam a posição e tamanho do retângulo para a elipse.


```
1  // Figura 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
```

Figura 5.26 | Desenhando uma cascata de formas com base na escolha do usuário. (Parte 1 de 2.)

```
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: // desenha elipses
31                   g.drawOval( 10 + i * 10, 10 + i * 10,
32                      50 + i * 10, 50 + i * 10 );
33                   break;
34             } // fim do switch
35          } // for final
36       } // fim do método paintComponent
37    } // fim da classe Shapes
```

Desenha um retângulo iniciando nas coordenadas *x*-*y* especificadas como os dois primeiros argumentos com a largura e a altura especificads pelos dois últimos argumentos

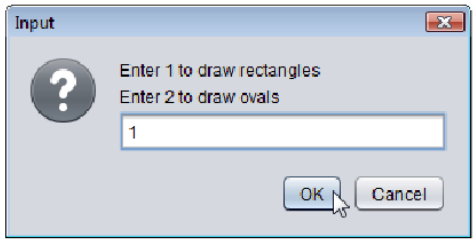
Desenha uma oval iniciando nas coordenadas *x*-*y* especificadas como os dois primeiros argumentos com a largura e a altura especificads pelos dois últimos argumentos

Figura 5.26 | Desenhando uma cascata de formas com base na escolha do usuário. (Parte 2 de 2.)


```
1  // Figura 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 ); // converte a entrada em 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
```

Figura 5.27 | Obtendo a entrada de usuário e criando um `JFrame` para exibir `Shapes`. (Parte 1 de 3.)

```
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
```



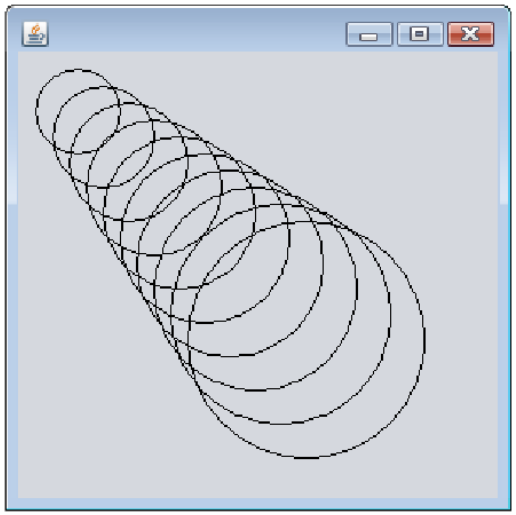
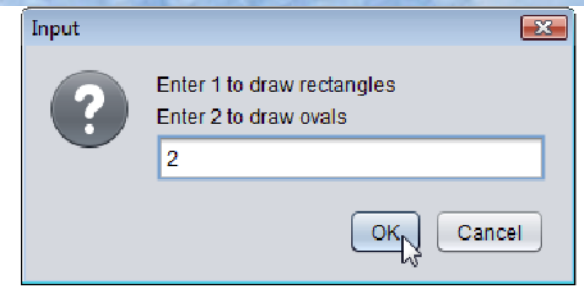


Figura 5.27 | Obtendo a entrada de usuário e criando um `JFrame` para exibir `Shapes`. (Parte 2 de 3.)



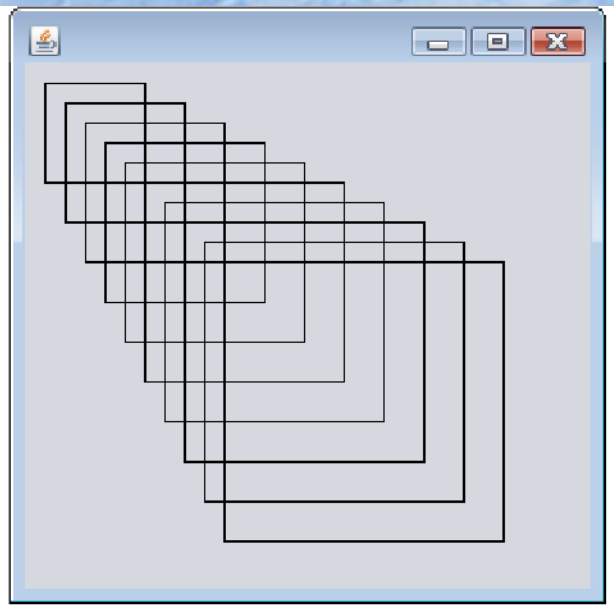


Figura 5.27 | Obtendo a entrada de usuário e criando um `JFrame` para exibir `Shapes`. (Parte 3 de 3.)