Chapter 5
Control Statements: Part 2

OBJECTIVES

In this Chapter you'll learn:

- The essentials of counter-controlled repetition.
- To use the `for` and `do...while` repetition statements to execute statements in a program repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` program control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.

5.1 Introduction

- `for` repetition statement
- `do...while` repetition statement
- `switch` multiple-selection statement
- `break` statement
- `continue` statement
- Logical operators
- Control statements summary.

5.2 Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - a **control variable** (or loop counter)
  - the **initial value** of the control variable
  - the **increment** (or **decrement**) by which the control variable is modified each time through the loop (also known as **each iteration of the loop**)
  - the **loop-continuation condition** that determines if looping should continue.

```
1  // Fig. 5.1: WhileCounter.java
2  // Counter-controlled repetition with the while repetition statement.
3
4  public class WhileCounter
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // declare and initialize control variable
9
10       while ( counter <= 10 ) // loop-continuation condition
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // increment control variable by 1
14       } // end while
15
16       System.out.println(); // output a newline
17    } // end main
18 } // end class WhileCounter
```

Declares and initializes control variable **counter** to 1

Loop-continuation condition tests for **count**'s final value

Initializes **gradeCounter** to 1; indicates first grade about to be input

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

Fig. 5.1 | Counter-controlled repetition with the `while` repetition statement.

Common Programming Error 5.1

Because floating-point values may be approximate, controlling loops with floating-point variables may result in imprecise counter values and inaccurate termination tests.

Error-Prevention Tip 5.1

Use integers to control counting loops.

Software Engineering Observation 5.1

"Keep it simple" is good advice for most of the code you'll write.

5.3 for Repetition Statement

- **for repetition statement**

```
1  // Fig. 5.2: ForCounter.java
2  // Counter-controlled repetition with the for repetition statement.
3
4  public class ForCounter
5  {
6     public static void main( String[] args )
7     {
8        // for statement header includes initialization,
9        // loop-continuation condition and increment
10       for ( int counter = 1; counter <= 10; counter++ )
11          System.out.printf( "%d  ", counter );
12
13       System.out.println(); // output a newline
14    } // end main
15 } // end class ForCounter
```

for statement's header contains everything you need for counter-controlled repetition

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

Fig. 5.2 | Counter-controlled repetition with the `for` repetition statement.

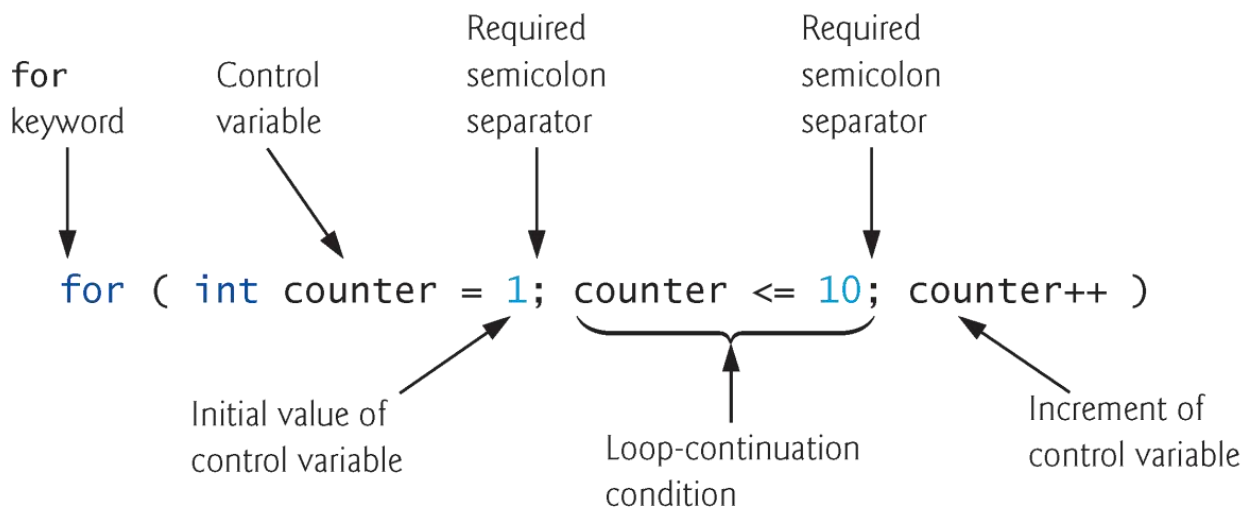


Fig. 5.3 | `for` statement header components.

5.3 `for` Repetition Statement (Cont.)

- The general format of the `for` statement is

  `for` (*initialization*; *loopContinuationCondition*; *increment*)
  statement

  - the *`initialization`* expression names the loop's control variable and optionally provides its initial value
  - *`loopContinuationCondition`* determines whether the loop should continue executing
  - *`increment`* modifies the control variable's value (possibly an increment or decrement), so that the loop-continuation condition eventually becomes false.
- The two semicolons in the `for` header are required.
- [redacted]ons in a `for` header are optional.

5.3 for Repetition Statement (Cont.)

- In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
initialization;
while ( loopContinuationCondition )
{
    statement
    increment;
}
```

- Typically, `for` statements are used for counter-controlled repetition and `while` statements for sentinel-controlled repetition.
- If the *initialization* expression in the `for` header declares the control variable, the control variable can be used only in that `for` statement.

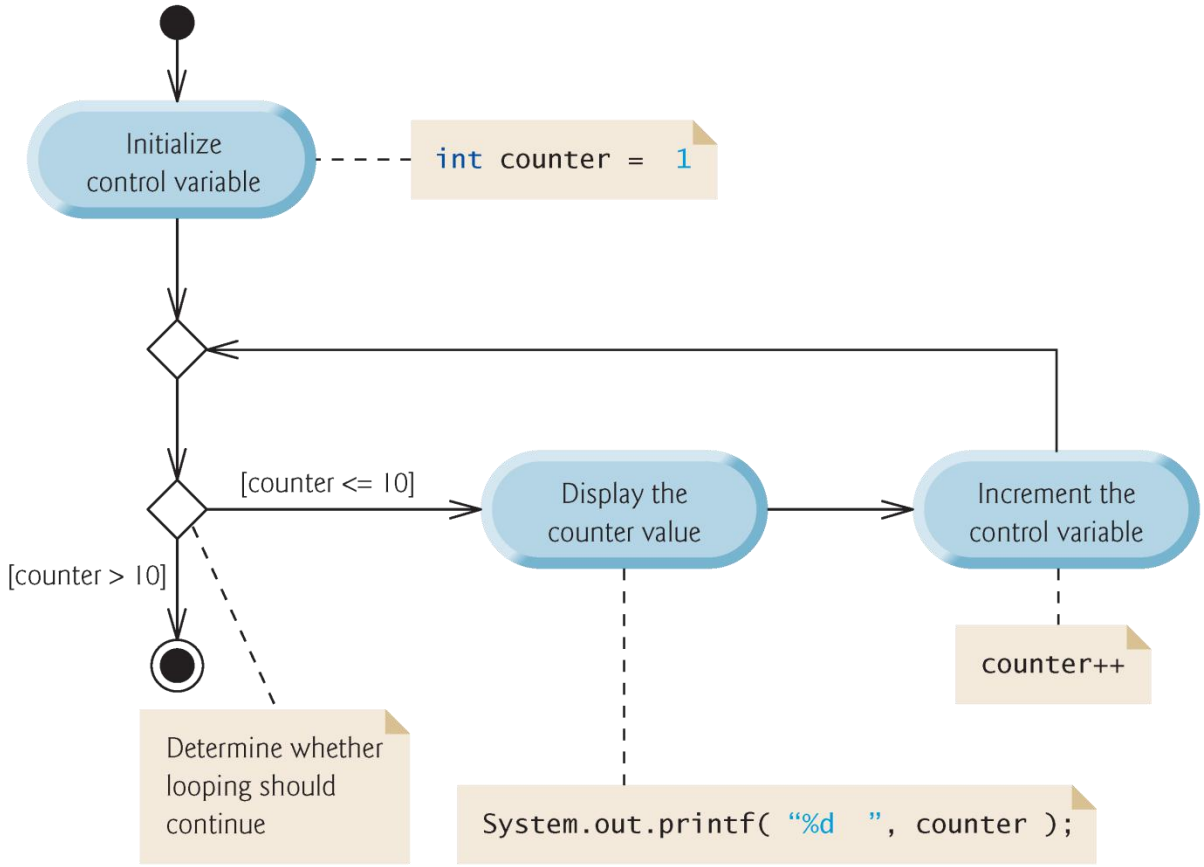


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.

```
1  // Fig. 5.5: Sum.java
2  // Summing integers with the for statement.
3
4  public class Sum
5  {
6     public static void main( String[] args )
7     {
8        int total = 0; // initialize total
9
10       // total even integers from 2 through 20
11       for ( int number = 2; number <= 20; number += 2 )
12          total += number;
13
14       System.out.printf( "Sum is %d\n", total ); // display results
15    } // end main
16 } // end class Sum
```

```
Sum is 110
```

Fig. 5.5 | Summing integers with the `for` statement.

5.3 for Repetition Statement (Cont.)

- Examples

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

for (int j = 2; j <= 80; j += 5) {…}

for ( int i = 100; i >= 1; i-- ){…}

for ( int number = 2;
   number <= 20;
   total += number, number += 2 )
   ; // empty statement
```

5.4 Examples Using the for Statement (Cont.)

- Compound interest application
  - *A person invests $1000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:*

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

    where

    p is the original amount invested (i.e., the principal)
    r is the annual interest rate (e.g., use 0.05 for 5%)
    n is the number of years
    a is the amount on deposit at the end of the nth year.

5.4 Examples Using the `for` Statement (Cont.)

- The solution to this problem (Fig. 5.6) involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.
- Java treats floating-point constants like `1000.0` and `0.05` as type `double`.
- Java treats whole-number constants like `7` and `-22` as type `int`.

```
1  // Fig. 5.6: Interest.java
2  // Compound-interest calculations with for.
3
4  public class Interest
5  {
6     public static void main( String[] args )
7     {
8        double amount; // amount on deposit at end of each year
9        double principal = 1000.0; // initial amount before interest
10       double rate = 0.05; // interest rate
11
12       // display headers
13       System.out.printf( "%s%20s\n", "Year", "Amount on deposit" );
14
15       // calculate amount on deposit for each of ten years
16       for ( int year = 1; year <= 10; year++ )
17       {
18          // calculate new amount for specified year
19          amount = principal * Math.pow( 1.0 + rate, year );
20
```

Java treats floating-point literals as `double` values

Uses `static` method `Math.pow` to help calculate the amount on deposit

Fig. 5.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

```
21            // display the year and the amount
22            System.out.printf( "%4d%,20.2f\n", year, amount );
23         } // end for
24      } // end main
25   } // end class Interest
```

Comma in format specifier indicates that large numbers should be displayed with thousands separators

```
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
```

Fig. 5.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

5.4 Examples Using the `for` Statement (Cont.)

- In the format specifier `%20s`
  - `20`:
    - `printf` displays the value with at least 20 character positions.
    - **right justified**
  - To indicate that values should be output **left justified**, precede the field width with the **minus sign (-) formatting flag** (e.g., `%-20s`).

5.4 Examples Using the `for` Statement (Cont.)

- Java does not include an exponentiation operator—`Math` class `static` method `pow` can be used for raising a value to a power.
- You can call a `static` method by specifying the class name followed by a dot (`.`) and the method name, as in
  - *ClassName*`.`*methodName*`(` *arguments* `)`
- `Math.pow(`*x*`,` *y*`)` calculates the value of x raised to the y^{th} power. The method receives two `double` arguments and returns a `double` value.

5.4 Examples Using the `for` Statement (Cont.)

- In the format specifier `%,20.2f`
  - **comma (,) formatting flag**
    - indicates that the floating-point value should be output with a **grouping separator.**
    - Separator is specific to the user's locale (i.e., country).
    - In the United States: 1,234.45.
  - **.2**
    - specifies the formatted number's precision—in this case, the number is rounded to the nearest hundredth and output with two digits to the right of

Performance Tip 5.1

In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop. Many of today's sophisticated optimizing compilers will place such calculations outside loops in the compiled code.

5.5 do...while Repetition Statement

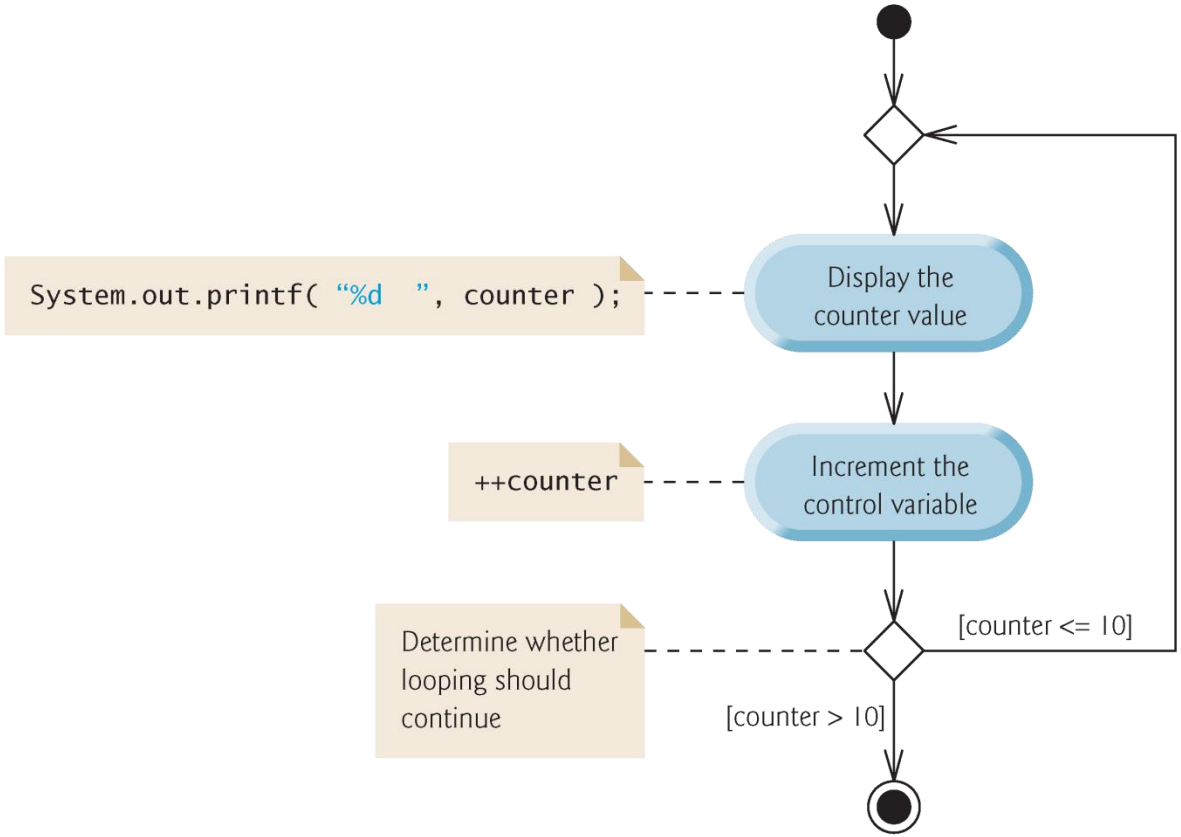


Fig. 5.8 nt UML activity diagram.

```
1  // Fig. 5.7: DoWhileTest.java
2  // do...while repetition statement.
3
4  public class DoWhileTest
5  {
6     public static void main( String[] args )
7     {
8        int counter = 1; // initialize counter
9
10       do
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter;
14       } while ( counter <= 10 ); // end do...while
15
16       System.out.println(); // outputs a newline
17    } // end main
18 } // end class DoWhileTest
```

Condition tested at end of loop, so loop always executes at least once

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

Fig. 5.7 | do...while repetition statement.

5.6 `switch` Multiple-Selection Statement

- **`switch` multiple-selection statement** performs different actions based on the possible values of a **constant integral expression** of type `byte`, `short`, `int` or `char`. As of Java SE 7, the expression may also be a `String`.

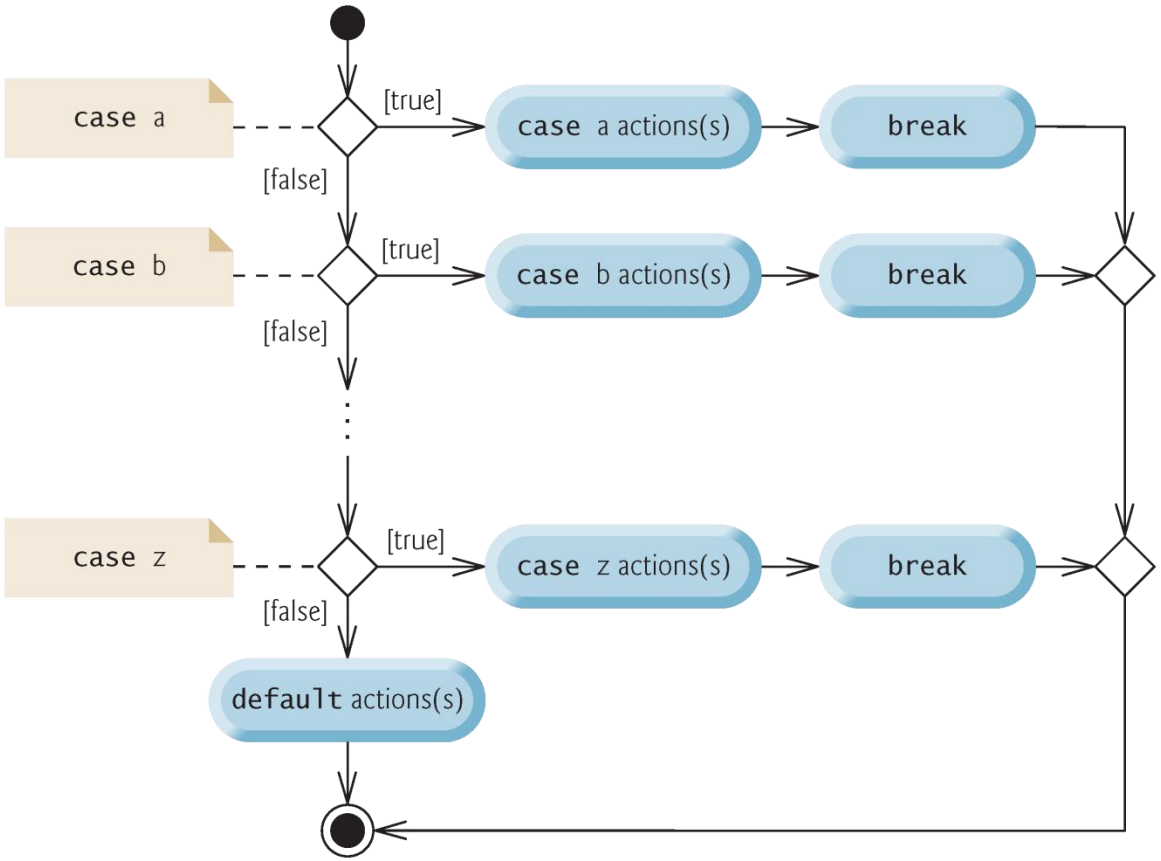


Fig. 5.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.

```
1  // Fig. 5.9: GradeBook.java
2  // GradeBook class uses switch statement to count letter grades.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class GradeBook
6  {
7     private String courseName; // name of course this GradeBook represents
8     // int instance variables are initialized to 0 by default
9     private int total; // sum of grades
10    private int gradeCounter; // number of grades entered
11    private int aCount; // count of A grades
12    private int bCount; // count of B grades
13    private int cCount; // count of C grades
14    private int dCount; // count of D grades
15    private int fCount; // count of F grades
16
17    // constructor initializes courseName;
18    public GradeBook( String name )
19    {
20       courseName = name; // initializes courseName
21    } // end constructor
22
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 1 of 7.)

```
23      // method to set the course name
24      public void setCourseName( String name )
25      {
26         courseName = name; // store the course name
27      } // end method setCourseName
28
29      // method to retrieve the course name
30      public String getCourseName()
31      {
32         return courseName;
33      } // end method getCourseName
34
35      // display a welcome message to the GradeBook user
36      public void displayMessage()
37      {
38         // getCourseName gets the name of the course
39         System.out.printf( "Welcome to the grade book for\n%s!\n\n",
40            getCourseName() );
41      } // end method displayMessage
42
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 2 of 7.)

```
43      // input arbitrary number of grades from user
44      public void inputGrades()
45      {
46         Scanner input = new Scanner( System.in );
47
48         int grade; // grade entered by user
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
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 3 of 7.)

```
56         // loop until user enters the end-of-file indicator
57         while ( input.hasNext() )
58         {
59            grade = input.nextInt(); // read grade
60            total += grade; // add grade to total
61            ++gradeCounter; // increment number of grades
62
63            // call method to increment appropriate counter
64            incrementLetterGradeCounter( grade );
65         } // end while
66      } // end method inputGrades
67
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 4 of 7.)

```
68      // add 1 to appropriate counter for specified grade
69      private void incrementLetterGradeCounter( int grade )
70      {
71         // determine which grade was entered
72         switch ( grade / 10 )
73         {
74            case 9:  // grade was between 90
75            case 10: // and 100, inclusive
76               ++aCount; // increment aCount
77               break; // necessary to exit switch
78
79            case 8: // grade was between 80 and 89
80               ++bCount; // increment bCount
81               break; // exit switch
82
83            case 7: // grade was between 70 and 79
84               ++cCount; // increment cCount
85               break; // exit switch
86
87            case 6: // grade was between 60 and 69
88               ++dCount; // increment dCount
89               break; // exit switch
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 5 of 7.)

```
90
91             default: // grade was less than 60
92                ++fCount; // increment fCount
93                break; // optional; will exit switch anyway
94             } // end switch
95          } // end method incrementLetterGradeCounter
96
97          // display a report based on the grades entered by the user
98          public void displayGradeReport()
99          {
100            System.out.println( "\nGrade Report:" );
101
102            // if user entered at least one grade...
103            if ( gradeCounter != 0 )
104            {
105               // calculate average of all grades entered
106               double average = (double) total / gradeCounter;
107
108               // output summary of results
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 );
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 6 of 7.)

```
112
113             "Number of students who received each grade:",
114             "A: ", aCount,   // display number of A grades
115             "B: ", bCount,   // display number of B grades
116             "C: ", cCount,   // display number of C grades
117             "D: ", dCount,   // display number of D grades
118             "F: ", fCount ); // display number of F grades
119       } // end if
120       else // no grades were entered, so output appropriate message
121          System.out.println( "No grades were entered" );
122    } // end method displayGradeReport
123 } // end class GradeBook
```

Fig. 5.9 | `GradeBook` class uses `switch` statement to count letter grades. (Part 7 of 7.)

Portability Tip 5.1

The keystroke combinations for entering end-of-file are system dependent.

Common Programming Error 5.6

Forgetting a `break` *statement when one is needed in a* `switch` *is a logic error.*

```
1  // Fig. 5.10: GradeBookTest.java
2  // Create GradeBook object, input grades and display grade report.
3
4  public class GradeBookTest
5  {
6     public static void main( String[] args )
7     {
8        // create GradeBook object myGradeBook and
9        // pass course name to constructor
10       GradeBook myGradeBook = new GradeBook(
11          "CS101 Introduction to Java Programming" );
12
13       myGradeBook.displayMessage(); // display welcome message
14       myGradeBook.inputGrades(); // read grades from user
15       myGradeBook.displayGradeReport(); // display report based on grades
16    } // end main
17 } // end class GradeBookTest
```

Fig. 5.10 | Create `GradeBook` object, input grades and display grade report. (Part 1 of 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

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80
```

Fig. 5.10 | Create **GradeBook** object, input grades and display grade report. (Part 2 of 3.)

```
Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 5.10 | Create `GradeBook` object, input grades and display grade report. (Part 3 of 3.)

5.6 `switch` Multiple-Selection Statement (Cont.)

- The **end-of-file indicator** is a system-dependent keystroke combination which the user enters to indicate that there is no more data to input.
- On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence
  - *<Ctrl> d*
- On Windows systems, end-of-file can be entered by typing
  - *<Ctrl> z*
- Windows typically displays the characters ^Z on the screen when [illegible]tor is typed.

5.6 `switch` Multiple-Selection Statement (Cont.)

- `Scanner` method `hasNext` determine whether there is more data to input. This method returns the `boolean` value `true` if there is more data; otherwise, it returns `false`.
- As long as the end-of-file indicator has not been typed, method `hasNext` will return `true`.

5.6 `switch` Multiple-Selection Statement (Cont.)

- If no match occurs between the controlling expression's value and a `case` label, the `default` case executes.
- If no match occurs and there is no `default` case, program control simply continues with the first statement after the `switch`.

Good Programming Practice 5.3

Although each `case` *and the* `default` *case in a* `switch` *can occur in any order, place the* `default` *case last. When the* `default` *case is listed last, the* `break` *for that case is not required.*

5.6 `switch` Multiple-Selection Statement (Cont.)

- you can use **character constants**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`—which represent the integer values of characters.
- The expression in each `case` can also be a **constant variable**—a variable that contains a value which does not change for the entire program. Such a variable is declared with keyword `final`.
- Java has a feature called enumerations. Enumeration constants can also be used in `case` labels.

5.7 Strings in switch Statements

```
53      // predicate method returns whether the state has no-fault insurance
54      public boolean isNoFaultState()
55      {
56         boolean noFaultState;
57
58         // determine whether state has no-fault auto insurance
59         switch (getState()) // get AutoPolicy object's state abbreviation
60         {
61            case "MA": case "NJ": case "NY": case "PA":
62               noFaultState = true;
63               break;
64            default:
65               noFaultState = false;
66               break;
67         }
68
69         return noFaultState;
70      }
```

```
         System.out.printf(
            "Account #: %d; Car: %s; State %s %s a no-fault state%n%n",
            policy.getAccountNumber(), policy.getMakeAndModel(),
            policy.getState(),
            (policy.isNoFaultState() ? "is": "is not"));
      }
```

jdk11 and earlier:

```
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
         System.out.println(6);
         break;
    case TUESDAY:
        System.out.println(7);
        break; case THURSDAY:
    case SATURDAY:
        System.out.println(8);
         break;
    case WEDNESDAY:
         System.out.println(9);
         break;
}
```

JDK12:

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY -> System.out.println(7);
    case THURSDAY, SATURDAY -> System.out.println(8);
    case WEDNESDAY -> System.out.println(9);
 }
```

JDK13:

```
static void howMany(int k) {
    System.out.println(
        switch (k) {
            case  1 -> "one"
            case  2 -> "two"
            default -> "many"
        }
    );
}
```

5.8 `break` and `continue` Statements

- The `break` statement, when executed in a `while`, `for`, `do`…`while` or `switch`, causes immediate exit from that statement.

```
1  // Fig. 5.13: BreakTest.java
2  // break statement exiting a for statement.
3  public class BreakTest
4  {
5     public static void main(String[] args)
6     {
7        int count; // control variable also used after loop terminates
8
9        for (count = 1; count <= 10; count++) // loop 10 times
10       {
11          if (count == 5)
12             break; // terminates loop if count is 5
13
14          System.out.printf("%d ", count);
15       }
16
17       System.out.printf("%nBroke out of loop at count = %d%n", count);
18    }
19 } // end class BreakTest
```

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

Fig. 5.13 | break statement exiting a for statement. (Part 2 of 2.)

5.8 `break` and `continue` Statements (Cont.)

- The `continue` statement, when executed in a `while`, `for` or `do…while`, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.

```
1  // Fig. 5.14: ContinueTest.java
2  // continue statement terminating an iteration of a for statement.
3  public class ContinueTest
4  {
5     public static void main(String[] args)
6     {
7        for (int count = 1; count <= 10; count++) // loop 10 times
8        {
9           if (count == 5)
10             continue; // skip remaining code in loop body if count is 5
11
12          System.out.printf("%d ", count);
13       }
14
15       System.out.printf("%nUsed continue to skip printing 5%n");
16    }
17 } // end class ContinueTest
```

Fig. 5.14 | continue statement terminating an iteration of a for statement. (Part 1 of 2.)

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

Fig. 5.14 | continue statement terminating an iteration of a for statement. (Part 2 of 2.)

5.9 Logical Operators

- Java's **logical operators** enable you to form more complex conditions by combining simple conditions.
- The logical operators are
  - && (conditional AND)
  - || (conditional OR)
  - & (boolean logical AND)
  - | (boolean logical inclusive OR)
  - ^ (boolean logical exclusive OR)
  - ! (logical NOT).
- [*Note:* The &, | and ^ operators are also bitwise operators when they are applied to integral operands.]

5.9 Logical Operators (Cont.)

- The parts of an expression containing && or || operators are evaluated only until it's known whether the condition is true or false. T
- This feature of conditional AND and conditional OR expressions is called **short-circuit evaluation.**

Common Programming Error 5.7

In expressions using operator &&*, a condition—we'll call this the dependent condition—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed* after *the other condition, or an error might occur. For example, in the expression* `( i != 0 ) && ( 10 / i == 2 )`*, the second condition must appear after the first condition, or a divide-by-zero error might occur.*

5.9 Logical Operators (Cont.)

- The %b **format specifier** displays the word “true” or the word “false” based on a `boolean` expression’s value.

```
1  // Fig. 5.18: LogicalOperators.java
2  // Logical operators.
3
4  public class LogicalOperators
5  {
6     public static void main( String[] args )
7     {
8        // create truth table for && (conditional AND) operator
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       // create truth table for || (conditional OR) operator
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
```

Value of each condition like this is displayed using format specifier %b

Fig. 5.18 | Logical operators. (Part 1 of 4.)

```
22          // create truth table for & (boolean logical AND) operator
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          // create truth table for | (boolean logical inclusive OR) operator
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          // create truth table for ^ (boolean logical exclusive OR) operator
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
```

Fig. 5.18 | Logical operators. (Part 2 of 4.)

```
45         // create truth table for ! (logical negation) operator
46         System.out.printf( "%s\n%s: %b\n%s: %b\n", "Logical NOT (!)",
47            "!false", ( !false ), "!true", ( !true ) );
48      } // end main
49   } // end class 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
```

Fig. 5.18 | Logical operators. (Part 3 of 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
```

Fig. 5.18 | Logical operators. (Part 4 of 4.)

Operators	Associativity	Type
`++ --`	right to left	unary postfix
`++ -- + - ! (`*type*`)`	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&`	left to right	boolean logical AND
`^`	left to right	boolean logical exclusive OR
`|`	left to right	boolean logical inclusive OR
`&&`	left to right	conditional AND
`||`	left to right	conditional OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment

Fig. 5.19 | Precedence/associativity of the operators discussed so far.

5.10 Structured Programming Summary

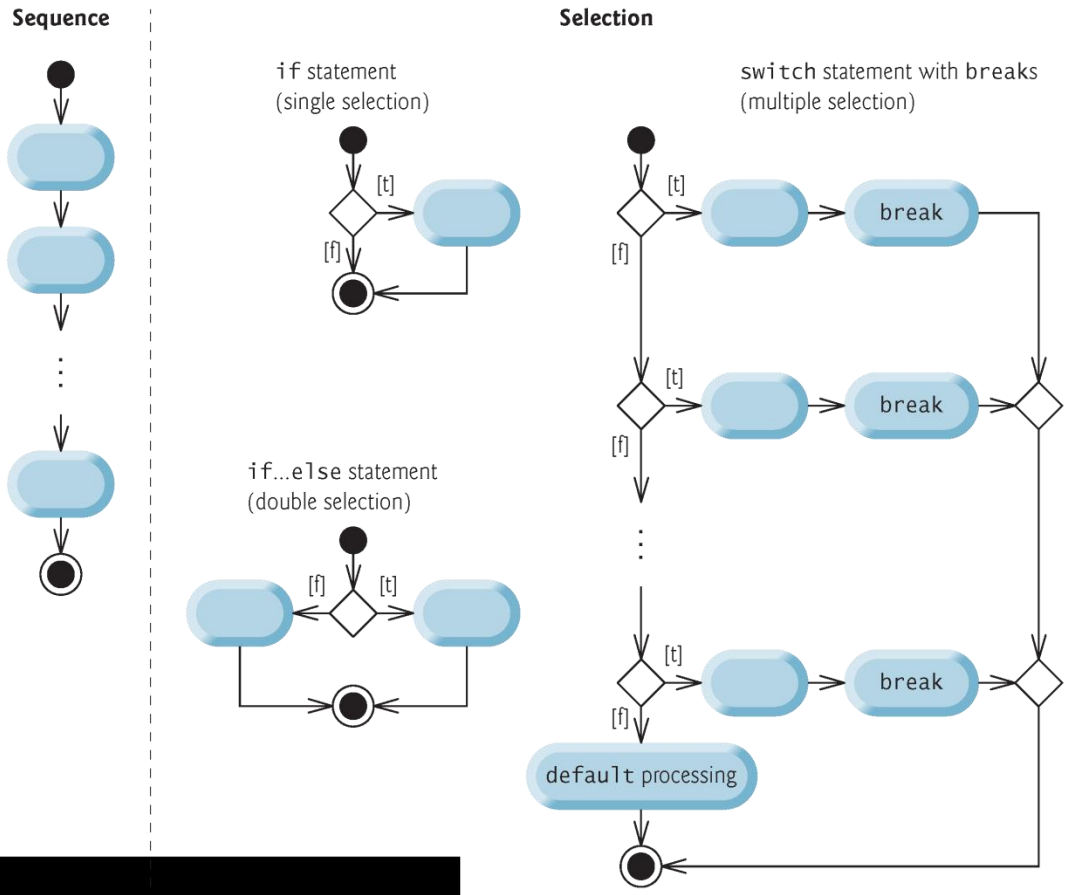


Fig. 5.20 | t sequence, selection and repetition statements. (Part 1 of 2.)

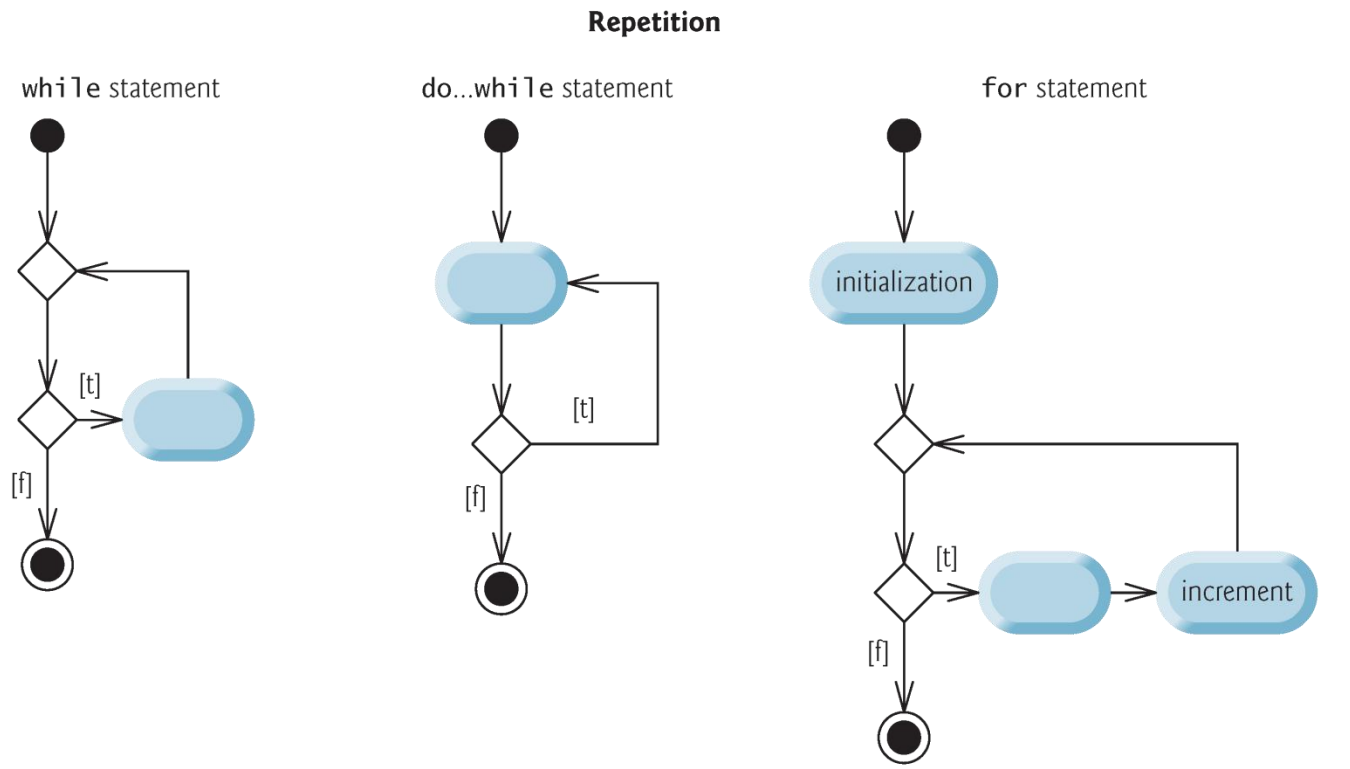


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

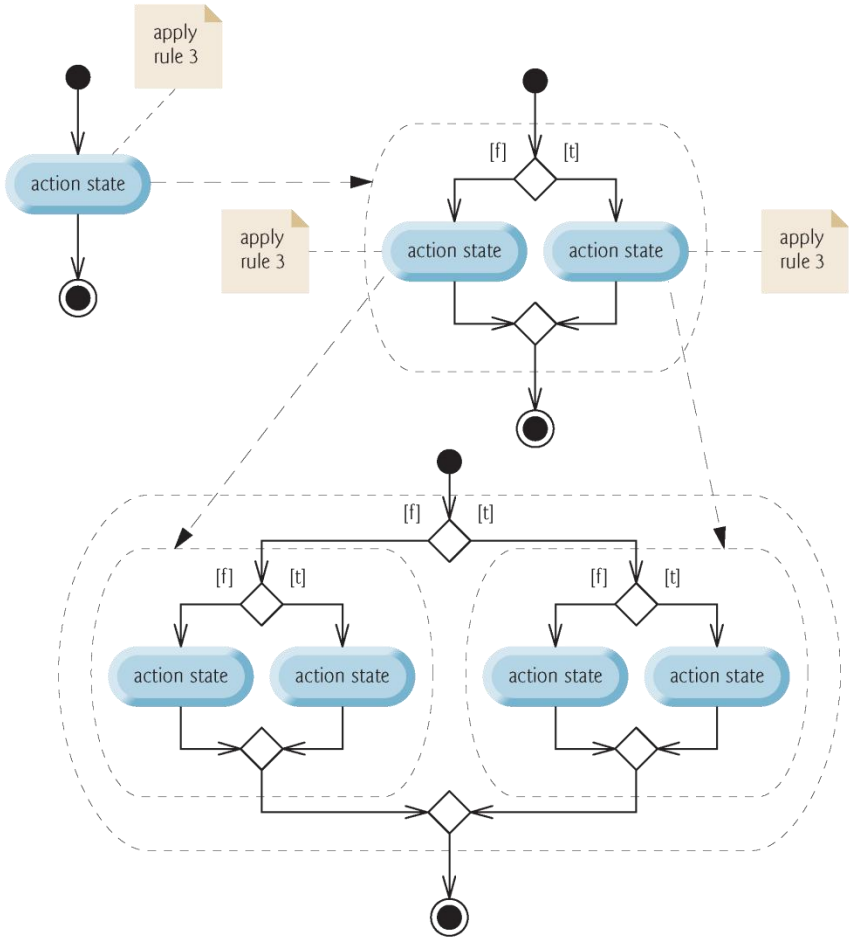
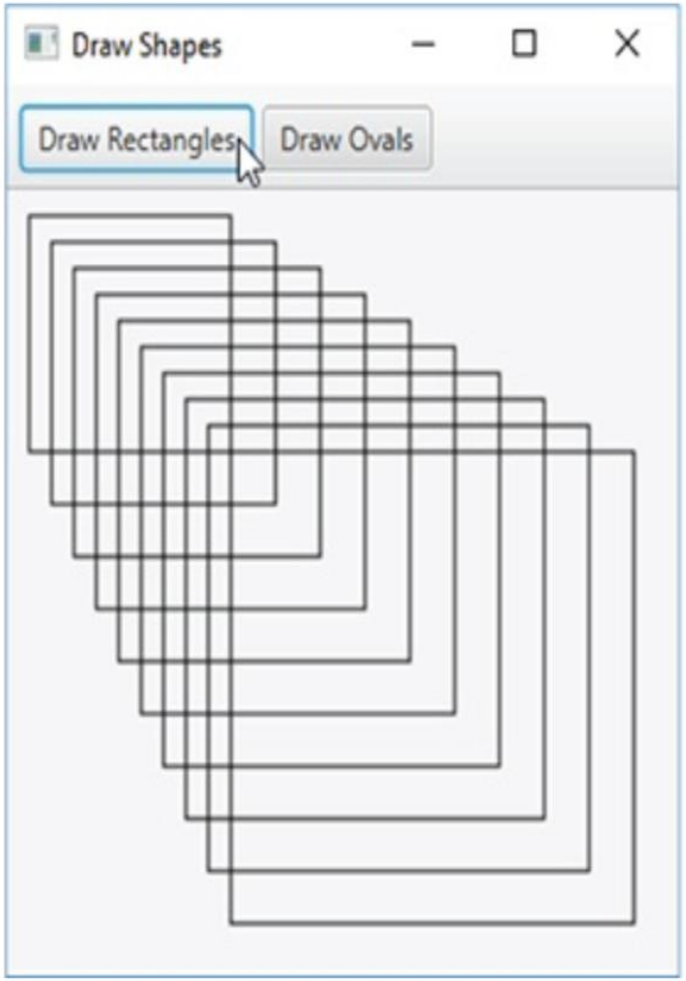
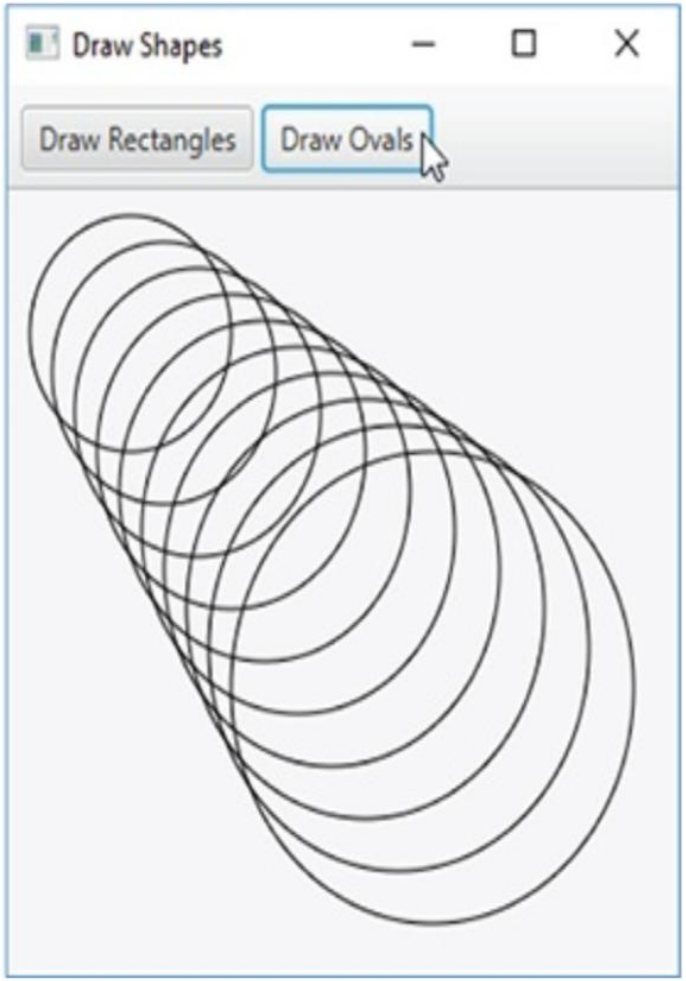


Fig. 5.24 e 3 of Fig. 5.21 to the simplest activity

5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals





5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals(Cont.)

- For the Button, set the following properties:
  - text——Draw Rectangle
  - On Action——drawRectanglesButtonPressed
- For the second Button, set the following properties:
  - text——Draw Oval
  - On Action—— drawOvalsButtonPressed

5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals (Cont.)

- As in the prior Java FX example, this app has two Java source-code files:
  - DrawShapes.java contains the JavaFX app's main application class that loads DrawShapes.fxml and configures the DrawShapesController.
  - DrawShapesController.java contains the controller class that draws either rectangles or ovals, based on which Button the user presses.

```java
package ch05.GUIGraphicsCaseStudy05;

// DrawShapes.java
// Main application class that loads and displays the DrawShapes GUI.
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class DrawShapes extends Application {
   @Override
   public void start(Stage stage) throws Exception {
      // loads Welcome.fxml and configures the DrawShapesController
      Parent root =
         FXMLLoader.load(getClass().getResource("DrawShapes.fxml"));

      Scene scene = new Scene(root); // attach scene graph to scene
      stage.setTitle("Draw Shapes"); // displayed in window's title bar
      stage.setScene(scene); // attach scene to stage
      stage.show(); // display the stage
   }

   public static void main(String[] args) {
      launch(args);
   }
}
```

```
1  package ch05.GUIGraphicsCaseStudy05;
2
3  // Fig. 3.13: DrawShapesController.java
4  // Using strokeRect and strokeOval to draw rectangles and ovals.
5  import javafx.event.ActionEvent;
6  import javafx.fxml.FXML;
7  import javafx.scene.canvas.Canvas;
8  import javafx.scene.canvas.GraphicsContext;
9
10 public class DrawShapesController {
11    @FXML private Canvas canvas;
12
13    // when user presses Draw Rectangles button, call draw for rectangles
14    @FXML
15    void drawRectanglesButtonPressed(ActionEvent event) {
16       draw("rectangles");
17    }
18
19    // when user presses Draw Ovals button, call draw for ovals
20    @FXML
21    void drawOvalsButtonPressed(ActionEvent event) {
22       draw("ovals");
23    }
24
```

```
24
25      // draws rectangles or ovals based on which Button the user pressed
26      public void draw(String choice) {
27         // get the GraphicsContext, which is used to draw on the Canvas
28         GraphicsContext gc = canvas.getGraphicsContext2D();
29
30         // clear the canvas for next set of shapes
31         gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
32
33         int step = 10;
34
35         // draw 10 overlapping shapes
36         for (int i = 0; i < 10; i++) {
37            // pick the shape based on the user's choice
38            switch (choice) {
39               case "rectangles": // draw rectangles
40                  gc.strokeRect(10 + i * step, 10 + i * step,
41                     90 + i * step, 90 + i * step);
42                  break;
43               case "ovals": // draw ovals
44                  gc.strokeOval(10 + i * step, 10 + i * step,
45                     90 + i * step, 90 + i * step);
46                  break;
47            }
48         }
49      }
```

5.11 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals (Cont.)

- Method clearRect of GraphicsContext clears any prior drawing
- Method strokeOval of GraphicsContext draws ovals.
- Mehtod strokeRect of GraphicsContext draws rectangles.