Chapter 5
Control Statements: Part 2

Java™ How to Program, 8/e

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.

5.3 `for` Repetition Statement

- **`for` repetition statement**
  - Specifies the counter-controlled-repetition details in a single line of code.
  - Figure 5.2 reimplements the application of Fig. 5.1 using `for`.

```
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.

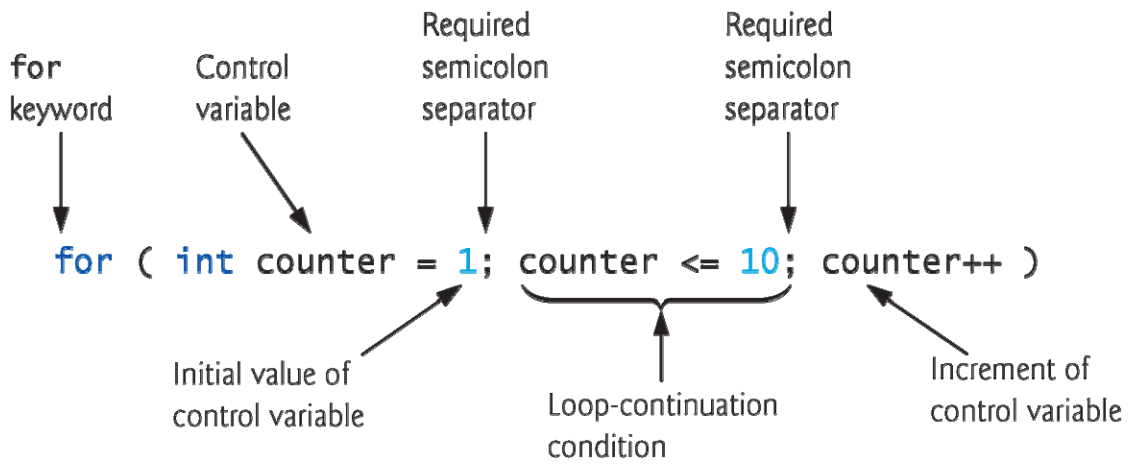


Fig. 5.3 | `for` statement header components.

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.

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.
- A variable's **scope** defines where it can be used in a program.
  - A local variable can be used only in the method that declares it and only from the point of declaration through the end of the method.

5.3 `for` Repetition Statement (Cont.)

- All three expressions in a `for` header are optional.
  - If the *loopContinuationCondition* is omitted, the condition is always true, thus creating an infinite loop.
  - You might omit the *initialization* expression if the program initializes the control variable before the loop.
  - You might omit the *increment* if the program calculates it with statements in the loop's body or if no increment is needed.
- The increment expression in a `for` acts as if it were a standalone statement at the end of the `for`'s body, so

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

  are equivalent increment expressions in a `for` statement.

5.3 `for` Repetition Statement (Cont.)

- The initialization, loop-continuation condition and increment can contain arithmetic expressions.
- For example, assume that `x = 2` and `y = 10`. If `x` and `y` are not modified in the body of the loop, the statement

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

- is equivalent to the statement

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

- The increment of a `for` statement may be negative, in which case it's a decrement, and the loop counts downward.

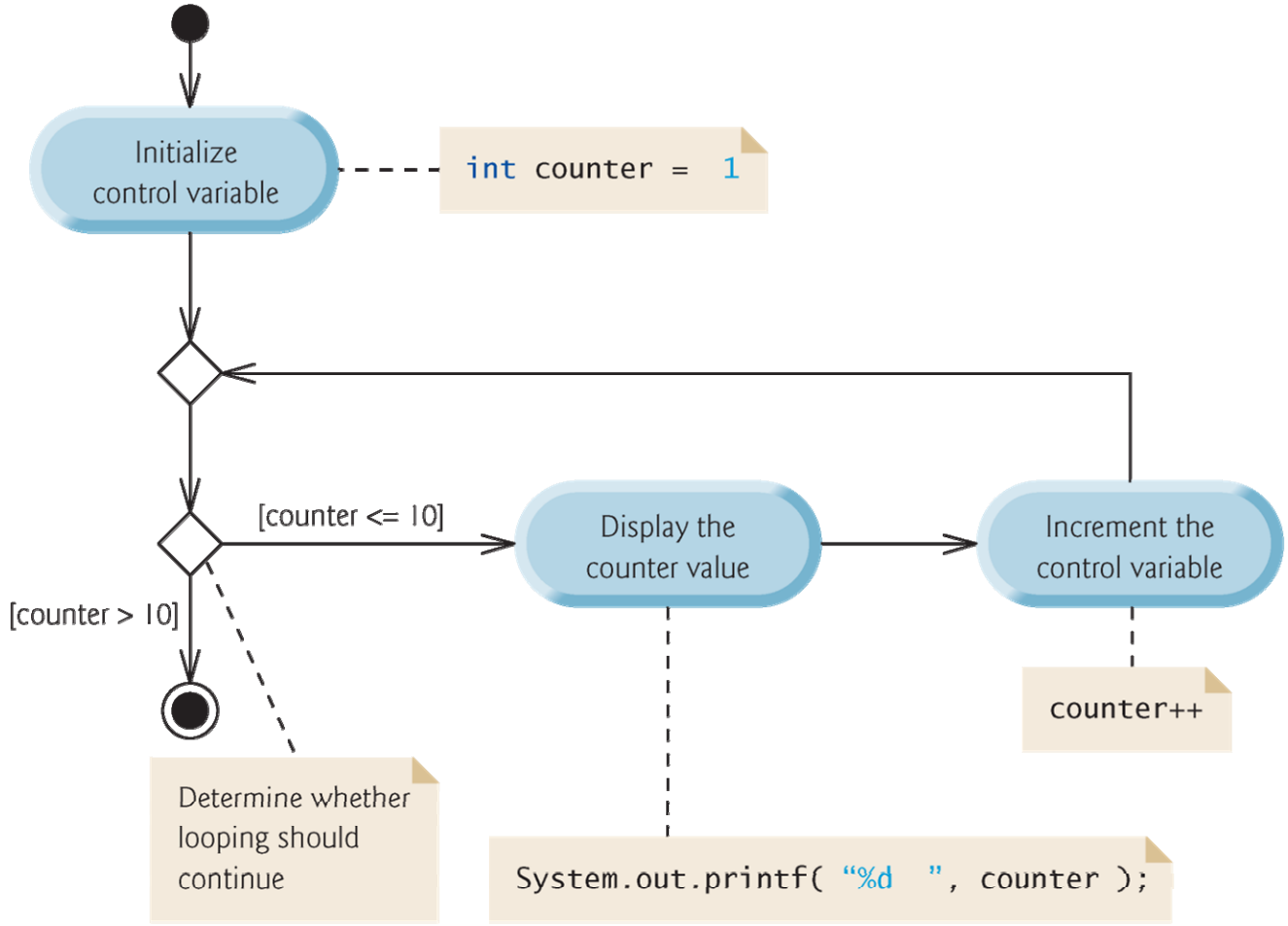


Fig. 5.4 | UML activity diagram for the `for` statement in Fig. 5.2.

5.4 Examples Using the `for` Statement

- a)Vary the control variable from `1` to `100` in increments of `1`.

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

- b)Vary the control variable from `100` to `1` in decrements of `1`.

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

- c)Vary the control variable from `7` to `77` in increments of `7`.

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

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

- d)Vary the control variable from 20 to 2 in decrements of 2.

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

- e)Vary the control variable over the values 2, 5, 8, 11, 14, 17, 20.

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

- f)Vary the control variable over the values 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

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

```
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
```

Note that the increment in this loop is 2, since we wish to total only the even integers

```
Sum is 110
```

Fig. 5.5 | Summing the even integers from 2 to 20 with the for 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`, the integer `20` between the `%` and the conversion character `s` indicates that the value output should be displayed with a **field width** of 20—that is, `printf` displays the value with at least 20 character positions.
- If the value to be output is less than 20 character positions wide, the value is **right justified** in the field by default.
- If the `year` value to be output has more characters than the field width, the field width would be extended to the right to accommodate the entire value.
- 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`, the **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, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,234.45.
- The number `20` in the format specification indicates that the value should be output right justified in a field width of 20 characters.
- The `.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 the decimal point.

5.5 do...while Repetition Statement

- The **do...while repetition statement** is similar to the `while` statement.
- In the `while`, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body; if the condition is false, the body never executes.
- The `do`...`while` statement tests the loop-continuation condition *after executing the loop's body*; therefore, the body always executes at least once.
- When a `do`...`while` statement terminates, execution continues with the next statement in sequence.

```
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.

5.5 do...while Repetition Statement (Cont.)

- Figure 5.8 contains the UML activity diagram for the `do...while` statement.
- The diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once.

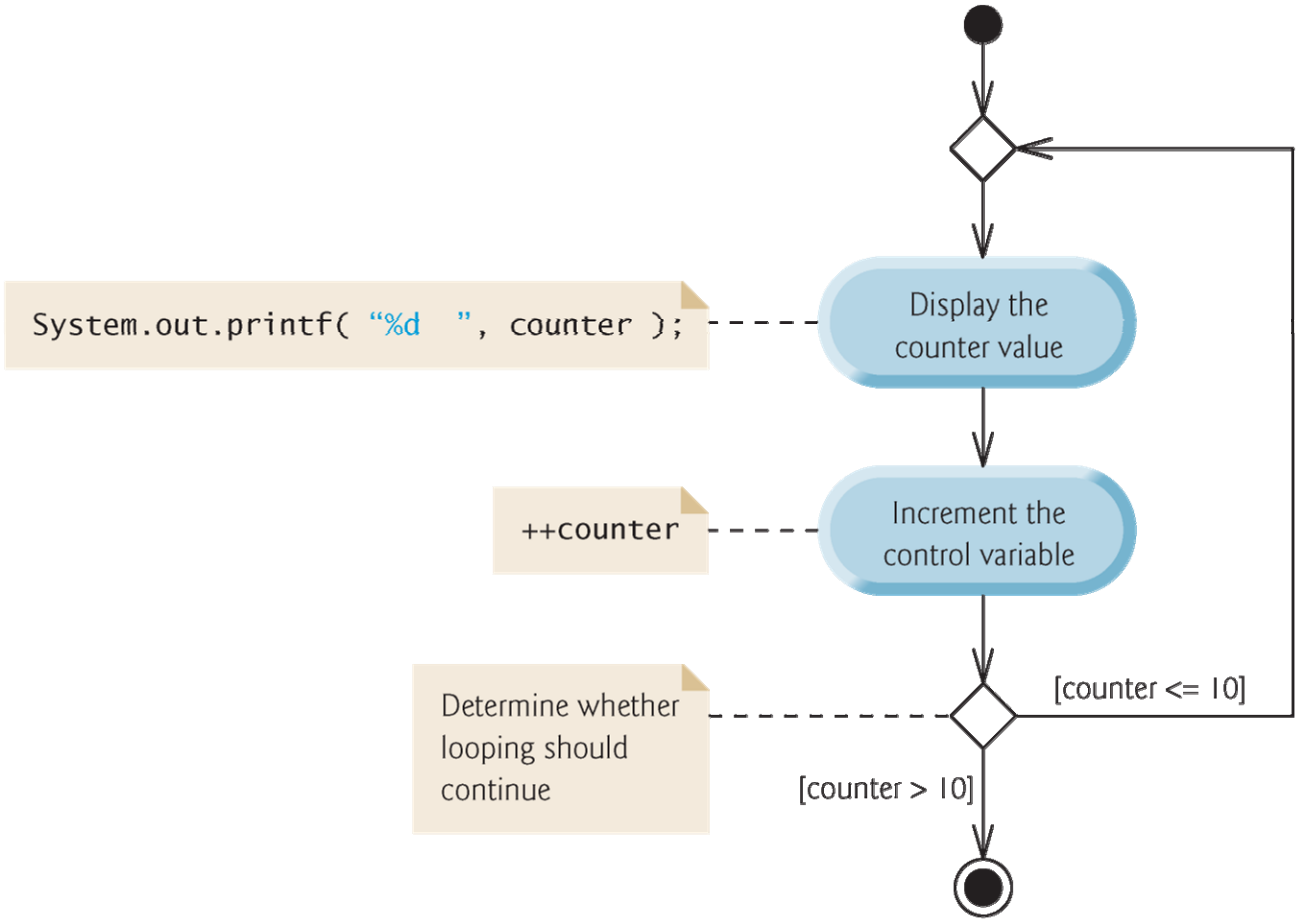


Fig. 5.8 | `do...while` repetition statement UML activity diagram.

5.5 do…while Repetition Statement (Cont.)

- Braces are not required in the `do…while` repetition statement if there's only one statement in the body.
- Most programmers include the braces, to avoid confusion between the `while` and `do…while` statements.
- Thus, the `do…while` statement with one body statement is usually written as follows:
  - ```
    do
    {
       statement
    } while ( condition );
    ```
    ```

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`.

```
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
```

No need to initialize these totals and counters to 0, though many programmers consider this a good programming practice

Fig. 5.9 | **GradeBook** class uses **switch** to count letter grades. (Part 1 of 6.)

```
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 to count letter grades. (Part 2 of 6.)

```
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
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
```

Loop continutes until end-of-file indicator is encountered

Fig. 5.9 | GradeBook class uses switch to count letter grades. (Part 3 of 6.)

```
67
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
90
```

grade/10 is the controlling expression; resulting integer value is compared to each case label's value

Fig. 5.9 | GradeBook class uses switch to count letter grades. (Part 4 of 6.)

```
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 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 );
```

`default` case executes for grades less than 60

Tests for possibility of division by zero

Fig. 5.9 | **GradeBook** class uses **switch** to count letter grades. (Part 5 of 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,    // 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** to count letter grades. (Part 6 of 6.)

```
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
```

Calling `GradeBook` `public` methods to input and summarize grades, then display grade report

Fig. 5.10 | `GradeBookTest` creates a `GradeBook` object and invokes its methods. (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
```

Fig. 5.10 | **GradeBookTest** creates a **GradeBook** object and invokes its methods. (Part 2 of 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
```

Fig. 5.10 | `GradeBookTest` creates a `GradeBook` object and invokes its methods. (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 a line by itself. This notation means to simultaneously press both the *Ctrl* key and the *d* key.
- On Windows systems, end-of-file can be entered by typing
  - *<Ctrl> z*
- On some systems, you must press *Enter* after typing the end-of-file key sequence.
- Windows typically displays the characters ^Z on the screen when the end-of-file indicator 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.)

- The `switch` statement consists of a block that contains a sequence of `case` **labels** and an optional **`default` case.**
- The program evaluates the **controlling expression** in the parentheses following keyword `switch`.
- The program compares the controlling expression's value (which must evaluate to an integral value of type `byte`, `char`, `short` or `int`) with each `case` label.
- If a match occurs, the program executes that `case`'s statements.
- The `break` **statement** causes program control to proceed with the first statement after the `switch`.

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

- `switch` does not provide a mechanism for testing ranges of values—every value must be listed in a separate `case` label.
- Note that each `case` can have multiple statements.
- `switch` differs from other control statements in that it does not require braces around multiple statements in a `case`.
- Without `break`, the statements for a matching case and subsequent cases execute until a `break` or the end of the `switch` is encountered. This is called "falling through."
- 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`.

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

- Figure 5.11 shows the UML activity diagram for the general `switch` statement.
- Most `switch` statements use a `break` in each `case` to terminate the `switch` statement after processing the `case`.
- The `break` statement is not required for the `switch`'s last `case` (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`.

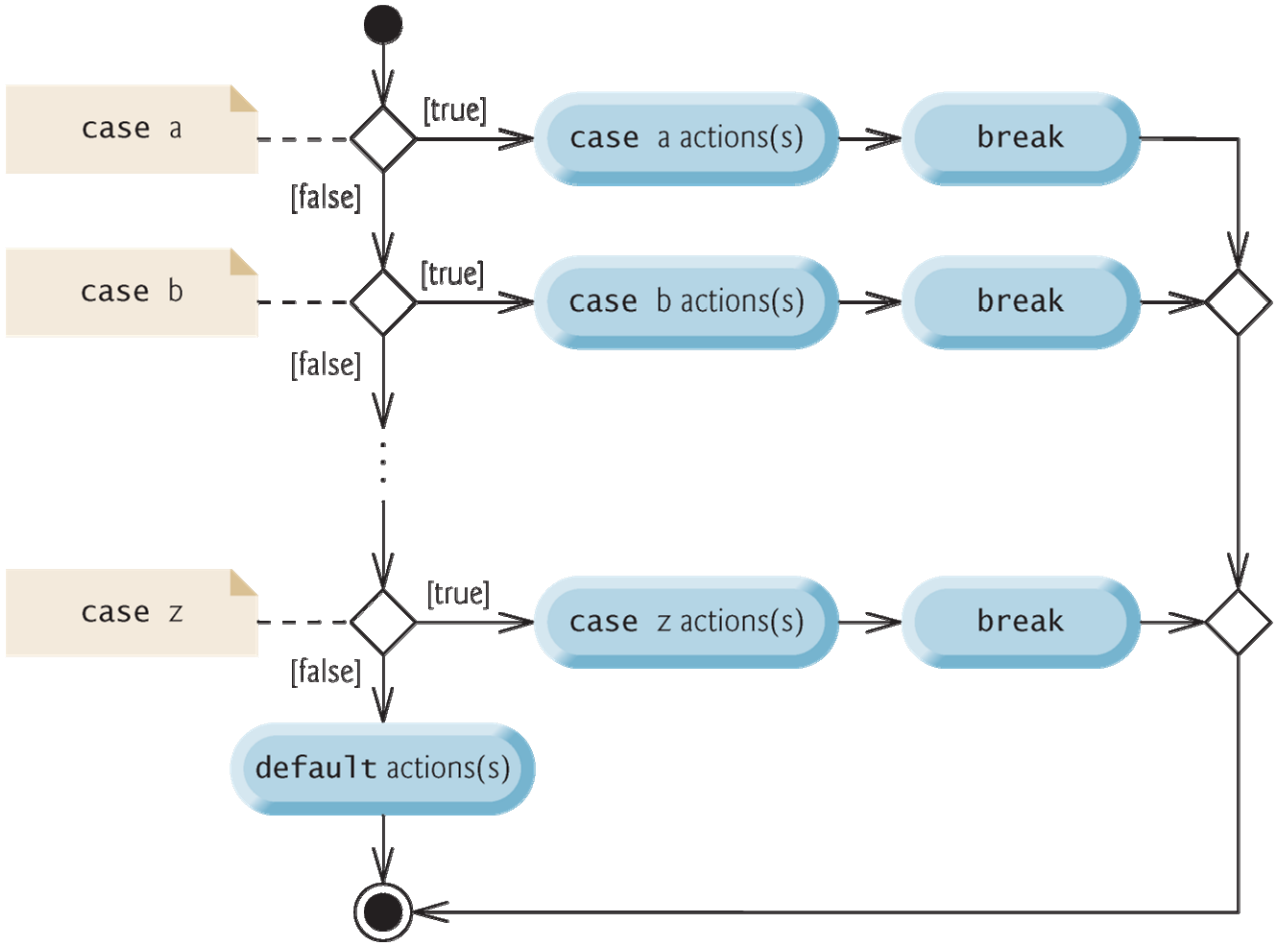


Fig. 5.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.

5.7 `break` and `continue` Statements

- The `break` statement, when executed in a `while`, `for`, `do`…`while` or `switch`, causes immediate exit from that statement.
- Execution continues with the first statement after the control statement.
- Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch`.

```
1  // Fig. 5.12: 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 ) // if count is 5,
12             break;         // terminate loop
13
14          System.out.printf( "%d ", count );
15       } // end for
16
17       System.out.printf( "\nBroke out of loop at count = %d\n", count );
18    } // end main
19 } // end class BreakTest
```

Terminates the loop immediately and program control continues at line 17

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

Fig. 5.12 | break statement exiting a for statement.

5.7 `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.
- In `while` and `do...while` statements, the program evaluates the loop-continuation test immediately after the `continue` statement executes.
- In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

```
1  // Fig. 5.13: 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 ) // if count is 5,
10             continue; // skip remaining code in loop
11
12          System.out.printf( "%d ", count );
13       } // end for
14
15       System.out.println( "\nUsed continue to skip printing 5" );
16    } // end main
17 } // end class ContinueTest
```

Terminates current iteration of loop and proceeds to increment

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

Fig. 5.13 | `continue` statement terminating an iteration of a `for` statement.

5.8 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.]

5.8 Logical Operators (Cont.)

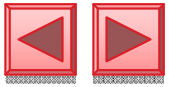
- The **&& (conditional AND)** operator ensures that two conditions are *both true* before choosing a certain path of execution.
- The table in Fig. 5.14 summarizes the `&&` operator. The table shows all four possible combinations of `false` and `true` values for *expression1* and *expression2*.
- Such tables are called **truth tables**. Java evaluates to `false` or `true` all expressions that include relational operators, equality operators or logical operators.

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.14 | && (conditional AND) operator truth table.

5.8 Logical Operators (Cont.)

- The || (**conditional OR)** operator ensures that *either or both* of two conditions are true before choosing a certain path of execution.
- Figure 5.15 is a truth table for operator conditional OR (||).
- Operator && has a higher precedence than operator ||.
- Both operators associate from left to right.



expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.15 | || (conditional OR) operator truth table.

5.8 Logical Operators (Cont.)

- The `!` (**logical NOT**, also called **logical negation** or **logical complement**) operator "reverses" the meaning of a condition.
- The logical negation operator is a unary operator that has only a single condition as an operand.
- The logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`.
- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator.
- Figure 5.17 is a truth table for the logical negation operator.

expression	!expression
false	true
true	false

Fig. 5.17 | ! (logical negation, or logical NOT) operator truth table.

5.8 Logical Operators (Cont.)

- Figure 5.18 produces the truth tables discussed in this section.
- 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.)

5.9 Structured Programming Summary

- Figure 5.20 uses UML activity diagrams to summarize Java's control statements.
- Java includes only single-entry/single-exit control statements—there is only one way to enter and only one way to exit each control statement.
- Connecting control statements in sequence to form structured programs is simple. The final state of one control statement is connected to the initial state of the next—that is, the control statements are placed one after another in a program in sequence. We call this control-statement stacking.
- The rules for forming structured programs also allow for control statements to be nested.

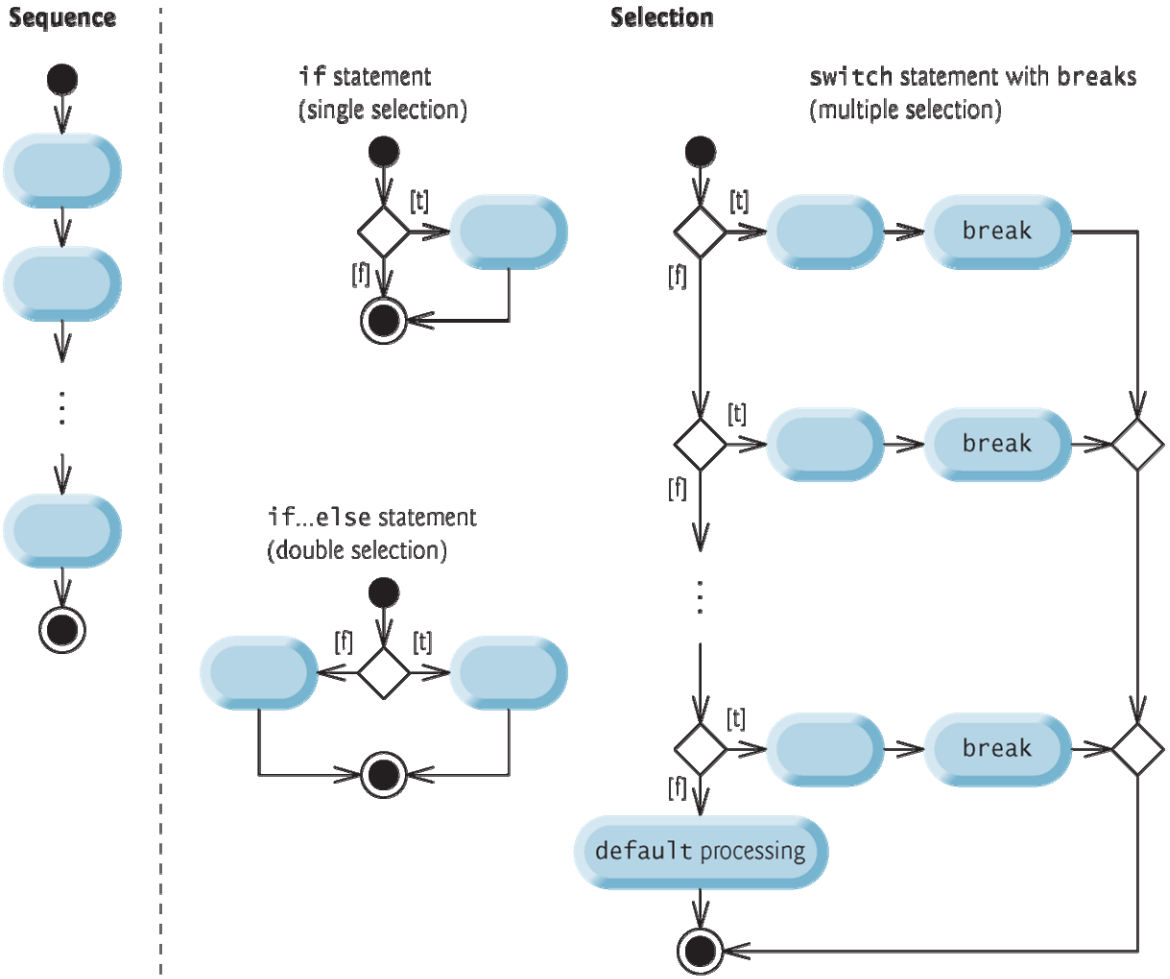


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 1 of 2.)

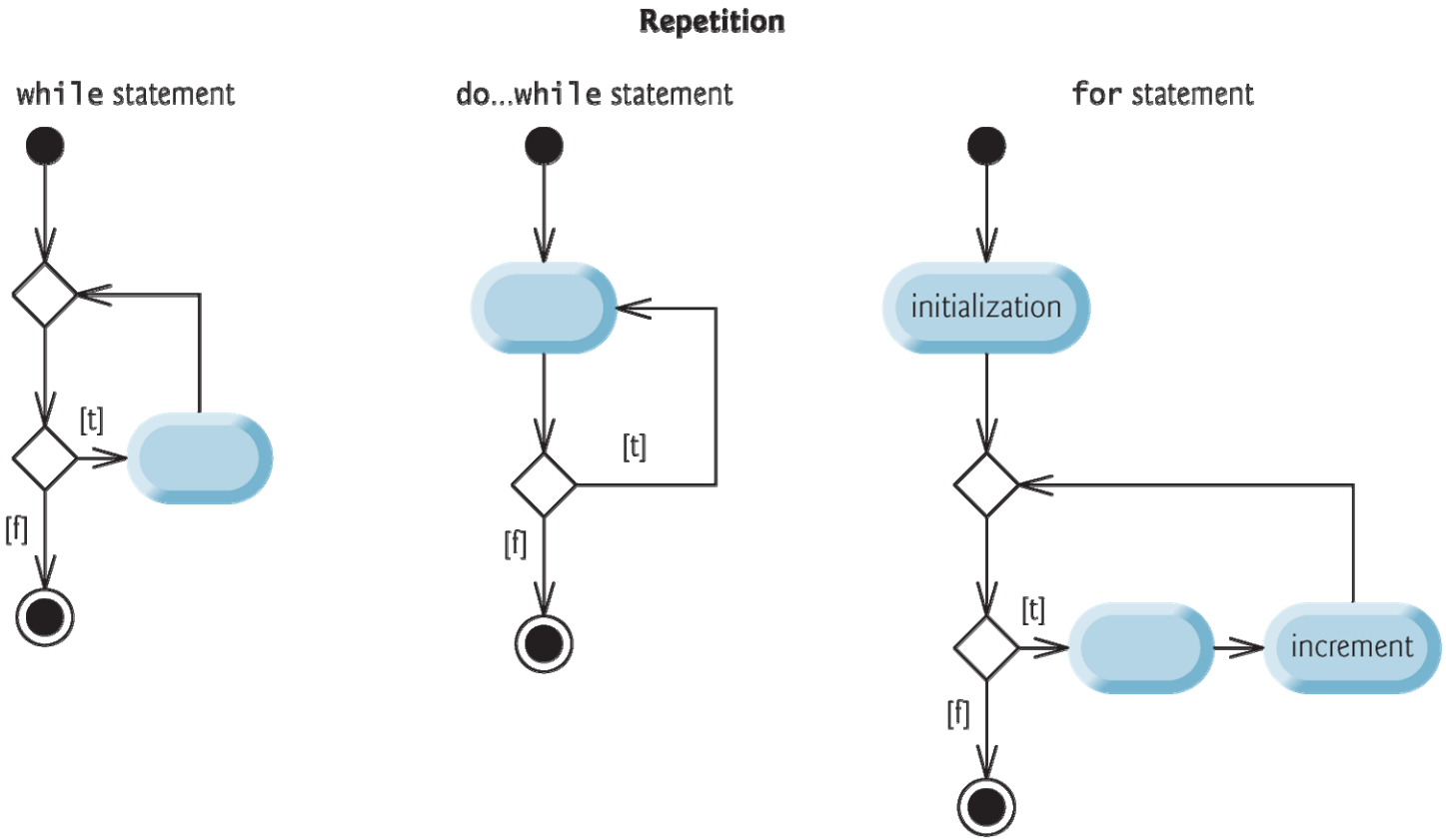


Fig. 5.20 | Java's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

Rules for forming structured programs

1. Begin with the simplest activity diagram (Fig. 5.22).
2. Any action state can be replaced by two action states in sequence. (This is the stacking rule.)
3. Any action state can be replaced by any control statement (sequence of action states, `if`, `if...else`, `switch`, `while`, `do...while` or `for`). (This is the nesting rule.)
4. Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 5.21 | Rules for forming structured programs.

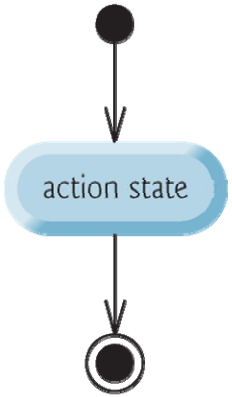


Fig. 5.22 | Simplest activity diagram.

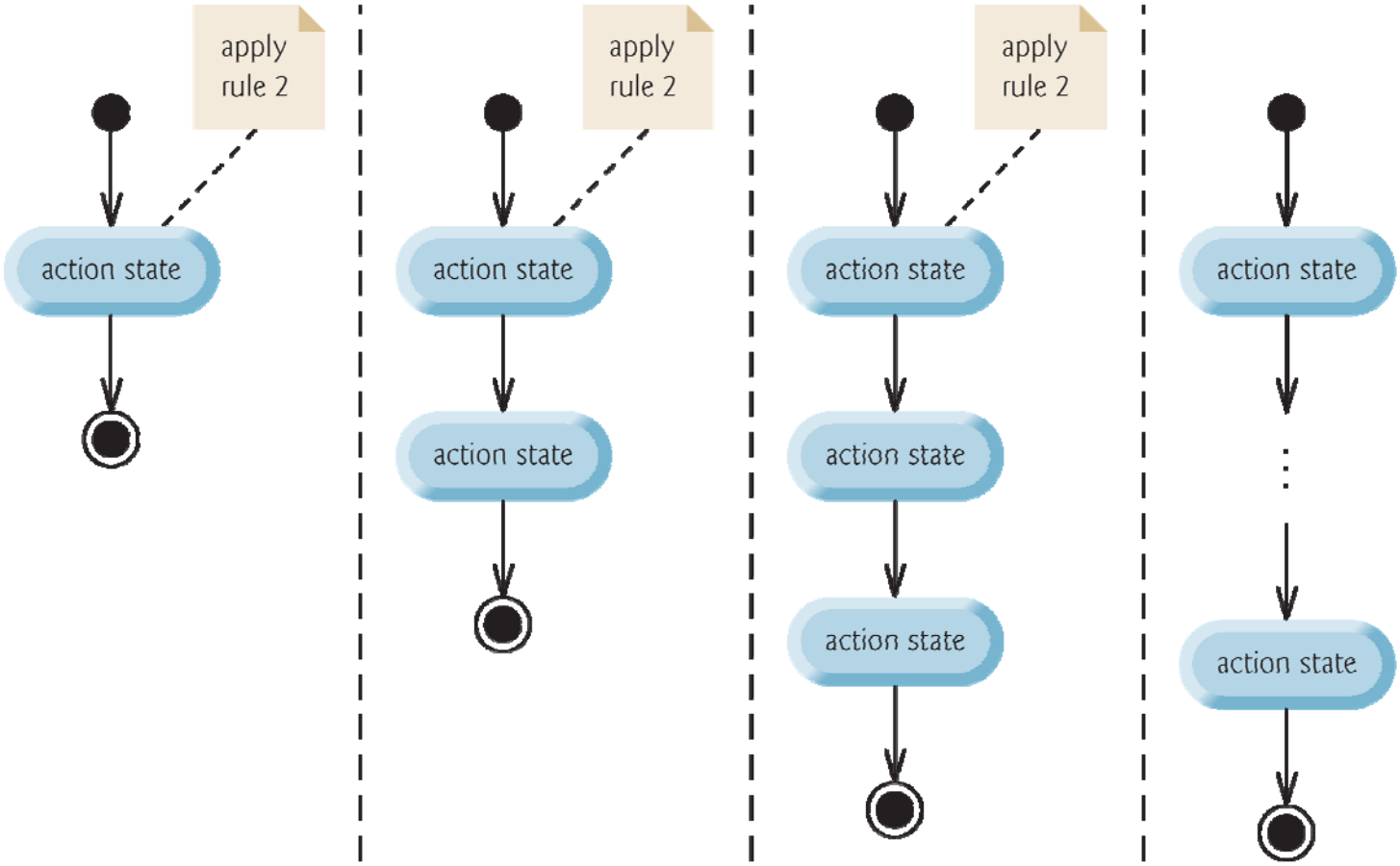


Fig. 5.23 | Repeatedly applying the stacking rule (rule 2) of Fig. 5.21 to the simplest activity diagram.

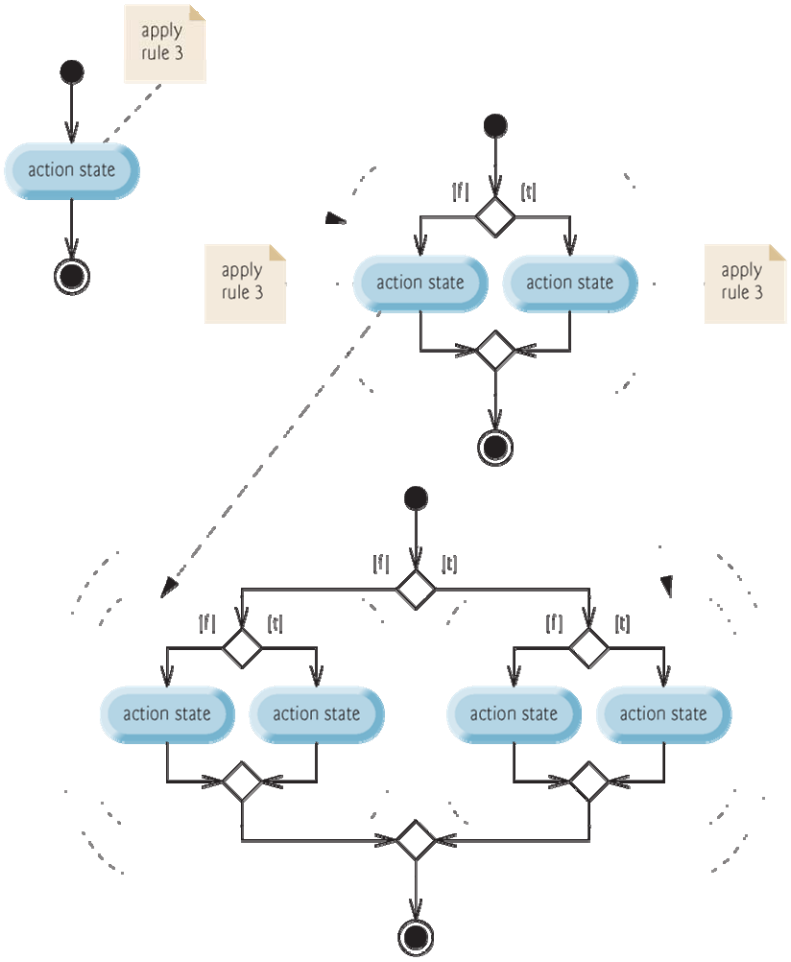


Fig. 5.24 | Repeatedly applying the nesting rule (rule 3) of Fig. 5.21 to the simplest activity diagram.

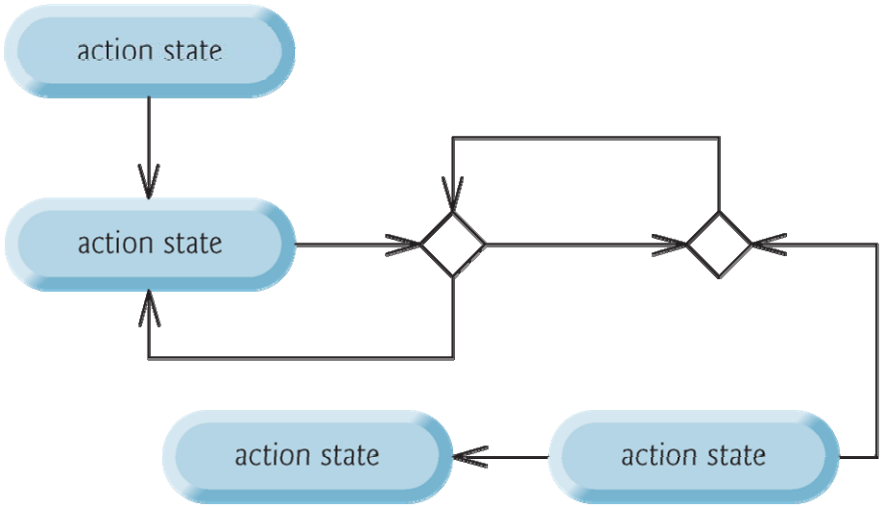


Fig. 5.25 | "Unstructured" activity diagram.

5.9 Structured Programming Summary (Cont.)

- Structured programming promotes simplicity.
- Bohm and Jacopini: Only three forms of control are needed to implement an algorithm:
  - Sequence
  - Selection
  - Repetition
- The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute.

5.9 Structured Programming Summary (Cont.)

- Selection is implemented in one of three ways:
  - `if` statement (single selection)
  - `if...else` statement (double selection)
  - `switch` statement (multiple selection)
- The simple `if` statement is sufficient to provide any form of selection—everything that can be done with the `if...else` statement and the `switch` statement can be implemented by combining `if` statements.

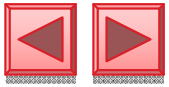
5.9 Structured Programming Summary (Cont.)

- Repetition is implemented in one of three ways:
  - `while` statement
  - `do...while` statement
  - `for` statement
- The `while` statement is sufficient to provide any form of repetition. Everything that can be done with `do...while` and `for` can be done with the `while` statement.

5.9 Structured Programming Summary (Cont.)

- Combining these results illustrates that any form of control ever needed in a Java program can be expressed in terms of
  - sequence
  - `if` statement (selection)
  - `while` statement (repetition)

  and that these can be combined in only two ways—stacking and nesting.



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

- `Graphics` methods `drawRect` and `drawOval`
- Method `drawRect` requires four arguments. The first two represent the *x*- and *y*-coordinates of the upper-left corner of the rectangle; the next two represent the rectangle's width and height.
- To draw an oval, method `drawOval` creates an imaginary rectangle called a **bounding rectangle** and places inside it an oval that touches the midpoints of all four sides.
- Method `drawOval` requires the same four arguments as method `drawRect`. The arguments specify the position and size of the bounding rectangle for the oval.

```
1  // Fig. 5.26: Shapes.java
2  // Demonstrates drawing different shapes.
3  import java.awt.Graphics;
4  import javax.swing.JPanel;
5
6  public class Shapes extends JPanel
7  {
8     private int choice; // user's choice of which shape to draw
9
10    // constructor sets the user's choice
11    public Shapes( int userChoice )
12    {
13       choice = userChoice;
14    } // end Shapes constructor
15
```

Fig. 5.26 | Drawing a cascade of shapes based on the user's choice. (Part 1 of 2.)

```
16      // draws a cascade of shapes starting from the top-left corner
17      public void paintComponent( Graphics g )
18      {
19         super.paintComponent( g );
20
21         for ( int i = 0; i < 10; i++ )
22         {
23            // pick the shape based on the user's choice
24            switch ( choice )
25            {
26               case 1: // draw rectangles
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            } // end switch
35         } // end for
36      } // end method paintComponent
37   } // end class Shapes
```

Draws a rectangle starting at the *x*-*y* coordinates specified as the first two arguments with the width and height specified by the last two arguments

Draws an oval in the bounding rectangle starting at the *x*-*y* coordinates specified as the first two arguments with the width and height specified by the last two arguments

Fig. 5.26 | Drawing a cascade of shapes based on the user's choice. (Part 2 of 2.)

```
1  // Fig. 5.27: ShapesTest.java
2  // Test application that displays class 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       // obtain user's choice
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       // create the panel with the user's input
18       Shapes panel = new Shapes( choice );
19
20       JFrame application = new JFrame(); // creates a new JFrame
21
```

Fig. 5.27 | Obtaining user input and creating a `JFrame` to display `Shapes`. (Part 1 of 3.)

```
22          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
23          application.add( panel ); // add the panel to the frame
24          application.setSize( 300, 300 ); // set the desired size
25          application.setVisible( true ); // show the frame
26       } // end main
27    } // end class ShapesTest
```

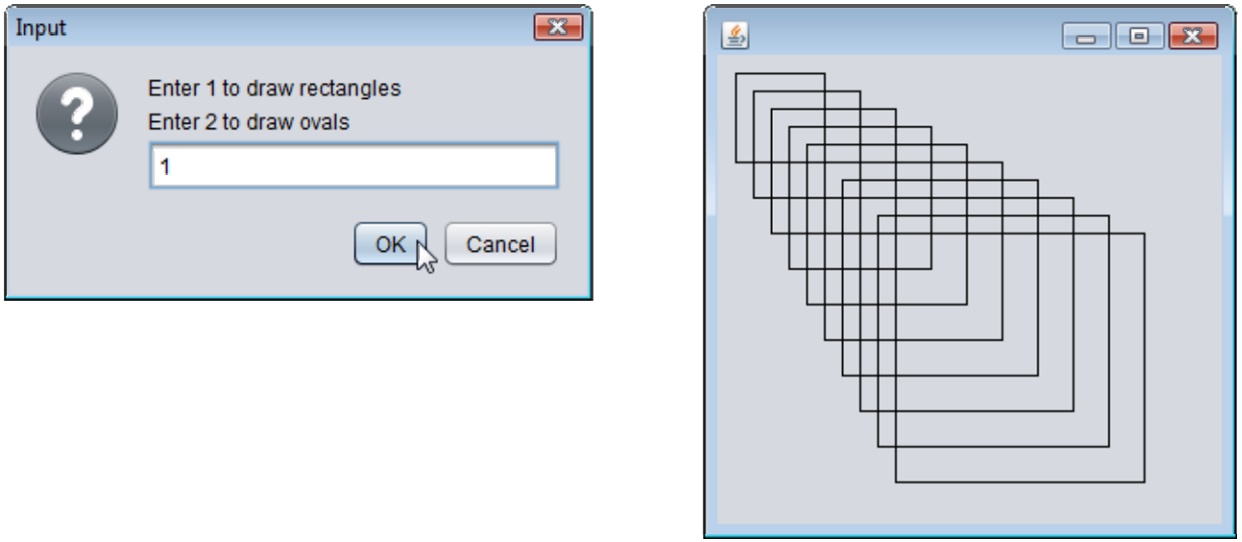


Fig. 5.27 | Obtaining user input and creating a `JFrame` to display `Shapes`. (Part 2 of 3.)

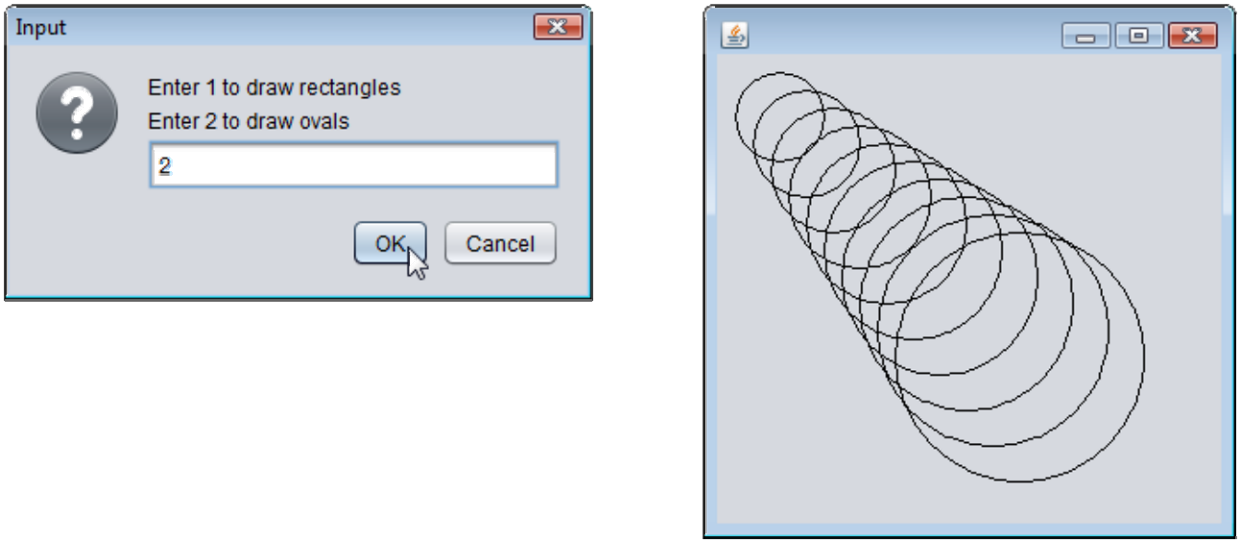


Fig. 5.27 | Obtaining user input and creating a `JFrame` to display `Shapes`. (Part 3 of 3.)

Exercises

- Program that prints the product of the odd integers from 1 to 15.
- Program that prints 5 histograms with lengths determined by user.
- Program that calculates Pythagorean triples (until 500).
- Program that prints 5 groups of 3 lines, each containing 4 asterisks.
- Graphics
  - Draw circles

More Exercises (explained)

- Computer Assisted Instruction
  - Program generates multiplication problems
- Graphics
  - Program prints 5 histograms with lengths determined by user

More Exercises

- Program that converts Fahrenheit to Celsius and vice versa.
- Program that simulates tossing a coin and counts number of tails or head.

Assignment 4! ☺

- Program that simulates a calculator
  - Addition
  - Multiplication
  - Division
  - Subtraction
  - Power (for xy user is asked x and y)
  - Sin
  - Cos
  - Log
- Menu of choice
  - A particular number for each menu choice (-1 exit)
  - User inserts the numbers after selecting the operation
  - Check for incorrect operations: division by zero, Log(-1) etc.

Where and when to submit

- Zip the project into a .zip document
- Rename the file in “Name Surname Assignment 4.zip”
- Submit the file by email to marenglenbiba@unyt.edu.al.
- Mail Subject: Java Assignment 4 – Name Surname
- Deadline 08/11/2010 23:59.
- 1 point will be taken off if the above rules are not respected as written.
- 1 point off for each day of delay

End of class