2c

Control Statements: Part 2

OBJECTIVES

In this lecture you will 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.

Outline

2c.1 Introduction

- **Continue structured-programming discussion**
  - **Introduce Java's remaining control structures**
    - `for`, `do...while`, `switch`

2c.2 Essentials of Counter-Controlled Repetition

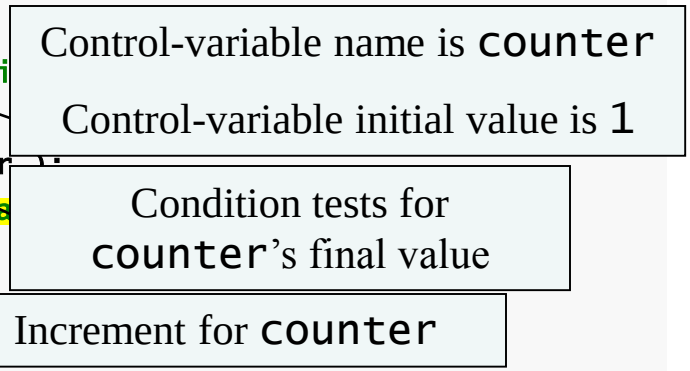
- **Counter-controlled repetition requires:**
  - **Control variable (loop counter)**
  - **Initial value of the control variable**
  - **Increment/decrement of control variable through each loop**
  - **Loop-continuation condition that tests for the final value of the control variable**

Outline

WhileCounter.java

```
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-conti
11       {
12          System.out.printf( "%d  ", counter );
13          ++counter; // increment control va
14       } // end while
15
16       System.out.println(); // output a
17    } // end main
18 } // end class WhileCounter
```

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



Common Programming Error 2c.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 2c.1

Control counting loops with integers.

Good Programming Practice 2c.1

Place blank lines above and below repetition and selection control statements, and indent the statement bodies to enhance readability.

Software Engineering Observation 2c.1

“Keep it simple” remains good advice for most of the code you will write.

2c.3 for Repetition Statement

- **Handles counter-controlled-repetition details**

Outline

```
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 Fo
```

```
1  2  3  4  5  6  7
```

Control-variable name is `counter`

Control-variable initial value is `1`

Increment for `counter`

Condition tests for `counter`'s final value

ForCounter.java

```
Line 10
int counter = 1;

Line 10
counter <= 10;

Line 10
counter++;
```

Common Programming Error 2c.2

Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of a repetition statement can cause an off-by-one error.

Good Programming Practice 2c.2

Using the final value in the condition of a `while` or for statement and using the <= relational operator helps avoid off-by-one errors. For a loop that prints the values 1 to 10, the loop-continuation condition should be `counter <= 10` rather than `counter < 10` (which causes an off-by-one error) or `counter < 11` (which is correct). Many programmers prefer so-called zero-based counting, in which to count 10 times, `counter` would be initialized to zero and the loop-continuation test would be `counter < 10`.

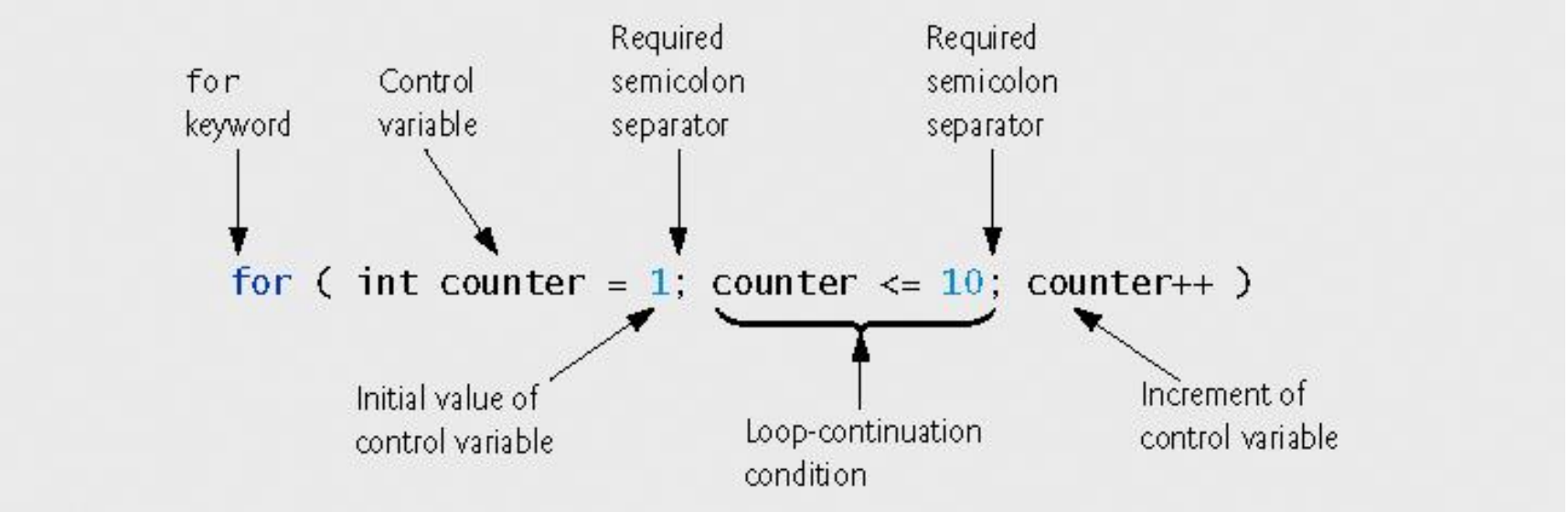


Fig. 2c.3 | for statement header components.

2c.3 `for` Repetition Statement (Cont.)

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

can usually be rewritten as:

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

Common Programming Error 2c.3

Using commas instead of the two required semicolons in a `for` header is a syntax error.

Common Programming Error 2c.4

When a `for` statement's control variable is declared in the initialization section of the `for`'s header, using the control variable after the `for`'s body is a compilation error.

Performance Tip 2c.1

There is a slight performance advantage to preincrementing, but if you choose to postincrement because it seems more natural (as in a `for` header), optimizing compilers will generate Java bytecode that uses the more efficient form anyway.

Good Programming Practice 2c.3

In the most cases, preincrementing and postincrementing are both used to add 1 to a variable in a statement by itself. In these cases, the effect is exactly the same, except that preincrementing has a slight performance advantage. Given that the compiler typically optimizes your code to help you get the best performance, use the idiom with which you feel most comfortable in these situations.

Common Programming Error 2c.5

Placing a semicolon immediately to the right of the right parenthesis of a `for` header makes that `for`'s body an empty statement. This is normally a logic error.

Error-Prevention Tip 2c.2

Infinite loops occur when the loop-continuation condition in a repetition statement never becomes `false`. To prevent this situation in a counter-controlled loop, ensure that the control variable is incremented (or decremented) during each iteration of the loop. In a sentinel-controlled loop, ensure that the sentinel value is eventually input.

Error-Prevention Tip 2c.3

Although the value of the control variable can be changed in the body of a `for` loop, avoid doing so, because this practice can lead to subtle errors.

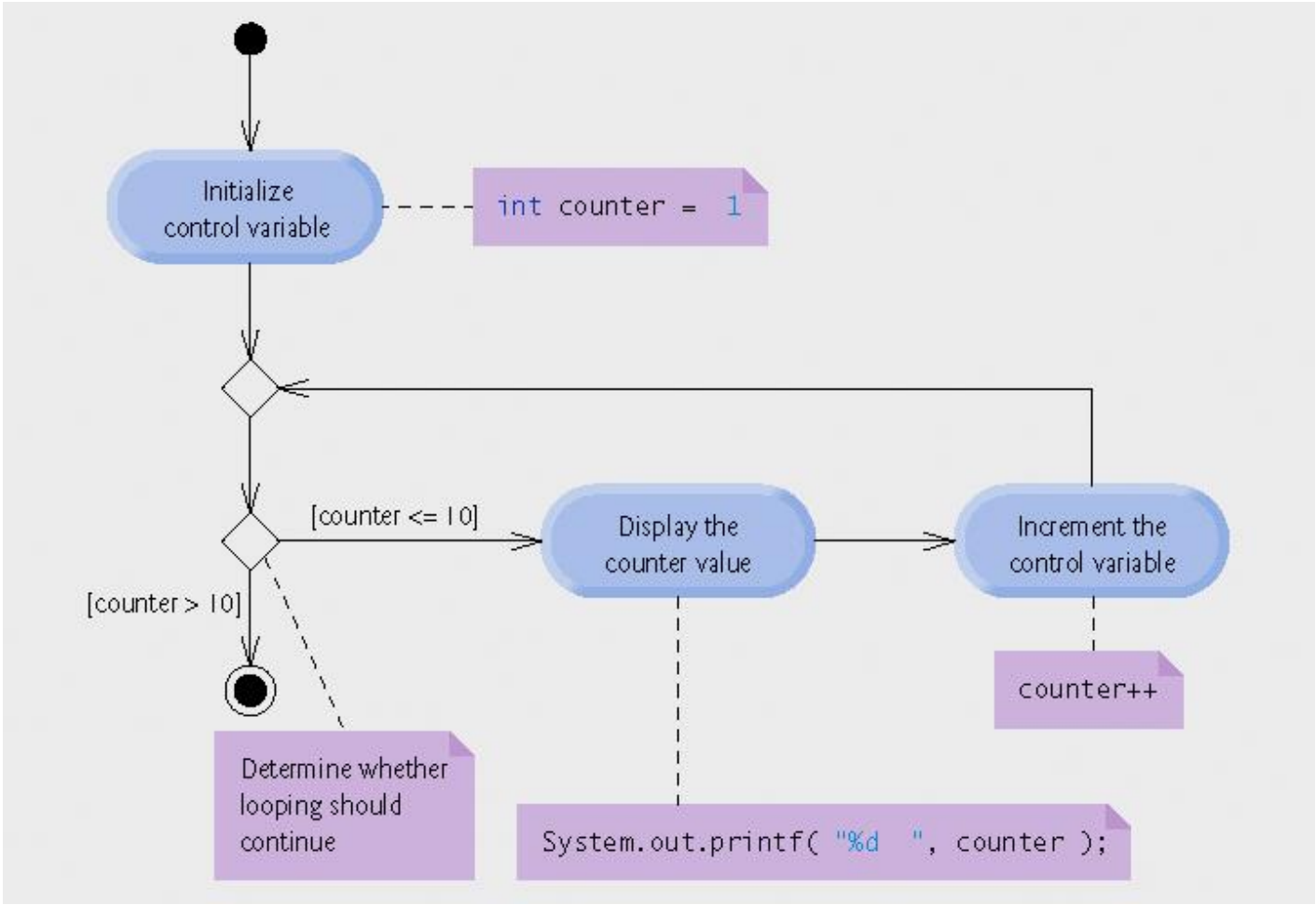


Fig. 2c.4 | UML activity diagram for the `for` statement in Fig. 2c.2.

2c.4 Examples Using the for Statement

- **Varying control variable in `for` statement**
  - Vary control variable from 1 to 100 in increments of 1
    - `for ( int i = 1; i <= 100; i++ )`
  - Vary control variable from 100 to 1 in increments of −1
    - `for ( int i = 100; i >= 1; i-- )`
  - Vary control variable from 7 to 77 in increments of 7
    - `for ( int i = 7; i <= 77; i += 7 )`
  - Vary control variable from 20 to 2 in decrements of 2
    - `for ( int i = 20; i >= 2; i -= 2 )`
  - Vary control variable over the sequence: 2, 5, 8, 11, 14, 17, 20
    - `for ( int i = 2; i <= 20; i += 3 )`
  - Vary control variable over the sequence: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0
    - `for ( int i = 99; i >= 0; i -= 11 )`

Common Programming Error 2c.6

Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` instead of `i >= 1` in a loop counting down to 1) is usually a logic error.

Outline

Sum.java

Line 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
```


increment `number` by 2 each iteration

```
Sum is 110
```

2c.4 Examples Using the for Statement (Cont.)

- **Initialization and increment expression can be comma-separated lists of expressions**
  - **E.g., lines 11-12 of Fig. 2c.5 can be rewritten as**

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

Good Programming Practice 2c.4

Limit the size of control statement headers to a single line if possible.

Good Programming Practice 2c.5

Place only expressions involving the control variables in the initialization and increment sections of a `for` statement. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the body of the loop (if they execute once per iteration of the loop, like increment or decrement statements).

Outline

Interest.java
(1 of 2)

Line 8

Line 13

```
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
```

Java treats literal values with decimal points as type `double`

Second string is right justified and displayed with a field width of 20

```
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
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
```

```
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
```

Calculate `amount` with `for` statement

Use the comma (`,`) formatting flag to display the amount with a thousands separator

Outline

`Interest.java`

(2 of 2)

Line 22

Program output

2c.4 Examples Using the `for` Statement (Cont.)

- **Formatting output**
  - **Field width**
  - **Minus sign (`-`) formatting flag for left justification**
  - **Comma (`,`) formatting flag to output numbers with grouping separators**
- **`static` method**
  - ***ClassName.methodName*(*arguments*)**

Good Programming Practice 2c.6

Do not use variables of type `double` (or `float`) to perform precise monetary calculations. The imprecision of floating-point numbers can cause errors that will result in incorrect monetary values. In the exercises, we explore the use of integers to perform monetary calculations. [*Note:* Some third-party vendors provide for-sale class libraries that perform precise monetary calculations. In addition, the Java API provides class `java.math.BigDecimal` for performing calculations with arbitrary precision floating-point values.]

Performance Tip 2c.2

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

2c.5 do…while Repetition Statement

- do…while statement
  - Similar to while statement
  - Tests loop-continuation after performing body of loop
    - i.e., loop body always executes at least once

Outline

```
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 cou
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
```

Declares and initializes control variable `counter`

Variable `counter`'s value is displayed before testing `counter`'s final value

DoWhileTest.java

Line 8

Lines 10-14

Program output

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

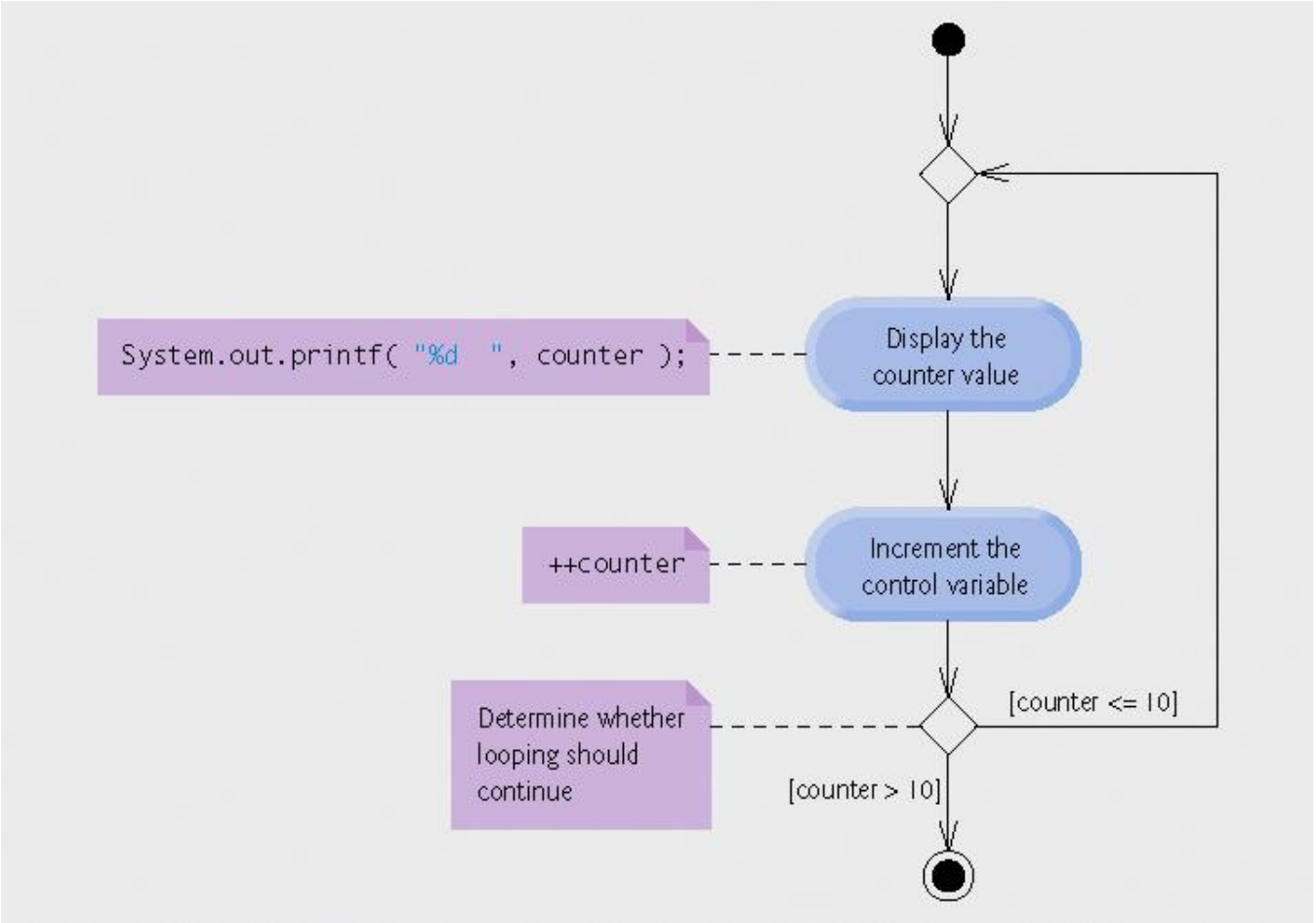


Fig. 2c.8 | `do...while` repetition statement UML activity diagram.

Good Programming Practice 2c.7

Always include braces in a `do...while` statement, even if they are not necessary. This helps eliminate ambiguity between the `while` statement and a `do...while` statement containing only one statement.

2c.6 switch Multiple-Selection Statement

- **switch statement**
  - **Used for multiple selections**

Outline

GradeBook.java

(1 of 5)

Lines 8-14

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

Outline

GradeBook.java

(2 of 5)

Lines 50-54

```
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
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
```

Display prompt

Outline

```
56      // loop until user enters the end-of-file indicator
57      while ( input.hasNext() )
58      {
59         grade = input.nextInt(); // read
60         total += grade; // add grade to t
61         ++gradeCounter; // increment numb
62
63         // call method to increment appropriate counter
64         incrementLetterGradeCounter( grade );
65      } // end while
66   } // end method inputGrades
67
68   // add 1 to appropriate counter for specified grade
69   public void incrementLetterGradeCounter( int numericGrade )
70   {
71      // determine which grade was entered
72      switch ( grade / 10 )
73      {
74         case 9:  // grade was between 90
75         case 10: // and 100
76            ++aCount; // increment aCount
77            break; // necessary to exit swi
78
79         case 8: // grade was between 80 and 89
80            ++bCount; // increment bCount
81            break; // exit switch
82
```

Loop condition uses method `hasNext` to determine whether there is more data to input

(`grade / 10`) is *controlling expression*

`switch` statement determines which `case` label to execute, depending on controlling expression

`GradeBook.java`

(3 of 5)

Line 57

Line 72 controlling expression

Lines 72-94

```
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
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
```

`default case` for grade less than 60

Outline

GradeBook.java

(4 of 5)

Line 91 default case

```
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 );
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
```

Outline

GradeBook.java

(5 of 5)

Portability Tip 2c.1

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

Common Programming Error 2c.7

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

Outline

GradeBookTest.java

```
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
```

(of 2)

Call `GradeBook` `public` methods to count grades

Lines 13-15

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

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

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

Outline

GradeBookTest.java

(2 of 2)

Program output

Software Engineering Observation 2c.2

Provide a `default` case in `switch` statements. Including a `default` case focuses you on the need to process exceptional conditions.

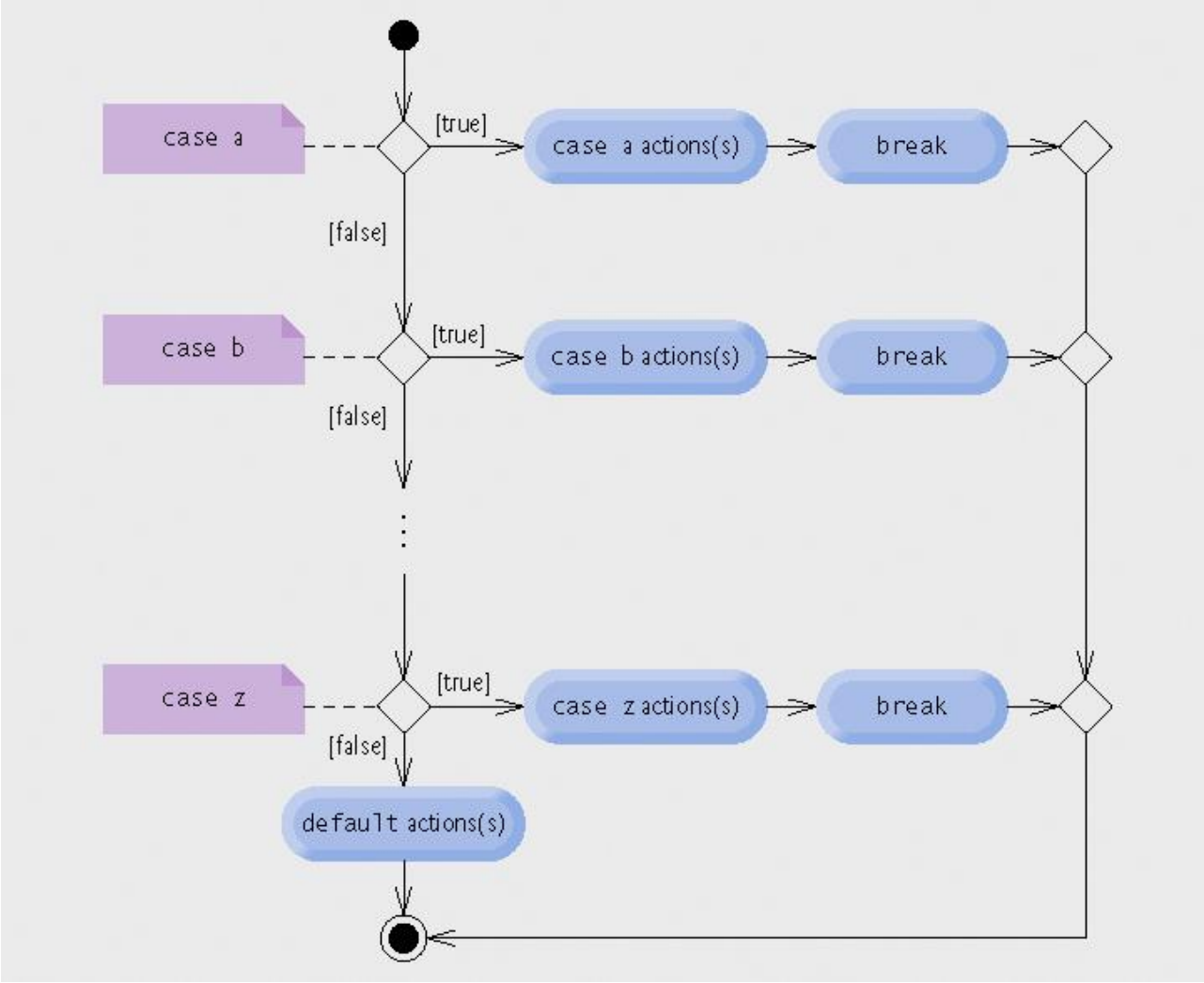


Fig. 2c.11 | `switch` multiple-selection statement UML activity diagram with `break` statements.

Good Programming Practice 2c.8

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. Some programmers include this `break` for clarity and symmetry with other cases.

2c.6 `switch` Multiple-Selection Statement (Cont.)

- **Expression in each `case`**
  - **Constant integral expression**
    - **Combination of integer constants that evaluates to a constant integer value**
  - **Character constant**
    - **E.g., ‘A’, ‘7’ or ‘$’**
  - **Constant variable**
    - **Declared with keyword `final`**

2c.7 break and continue Statements

- break/continue
  - Alter flow of control
- break statement
  - Causes immediate exit from control structure
    - Used in while, for, do…while or switch statements
- continue statement
  - Skips remaining statements in loop body
  - Proceeds to next iteration
    - Used in while, for or do…while statements

Strings in a *switch* command-

New in JDK 7

```
public static void main(String[] args) {
        String color = "red";
        String colorRGB = null;
        // Convert to switch over Strings
        if (color.equals("black")) {
            colorRGB = "008090";
        }
        else if(color.equals("red"))
            colorRGB = "00FF00";
        else {
            colorRGB = "00FFFF";
        }
 }
```

Strings in a *switch* command

```
public static void main(String[] args) {
        String color = "red";
        String colorRGB = null;
        switch (color) {
            case "black":
                colorRGB = "008090";
                break;
            case "red":
                colorRGB = "00FF00";
                break;
            default:
                colorRGB = "00FFFF";
                break;
        }
}
```

Outline

```
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
8
9        for ( count = 1; count <= 10; count++ ) //
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
```

Loop 10 times

Exit **for** statement (break) when count equals 5

BreakTest.java

Line 9

Lines 11-12

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

Program output

Outline

```
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++
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
```

Loop 10 times

Skip line 12 and proceed to line 7 when count equals 5

ContinueTest.java

Line 7

Lines 9-10

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

Program output

Software Engineering Observation 2c.3

Some programmers feel that `break` and `continue` violate structured programming. Since the same effects are achievable with structured programming techniques, these programmers do not use `break` or `continue`.

Software Engineering Observation 2c.4

There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.

2c.8 Logical Operators

- **Logical operators**
  - **Allows for forming more complex conditions**
  - **Combines simple conditions**
- **Java logical operators**
  - **&& (conditional AND)**
  - **|| (conditional OR)**
  - **& (boolean logical AND)**
  - **| (boolean logical inclusive OR)**
  - **^ (boolean logical exclusive OR)**
  - **! (logical NOT)**

2c.8 Logical Operators (Cont.)

- **Conditional AND (&&) Operator**
  - **Consider the following `if` statement**

```
if ( gender == FEMALE && age >= 65 )
   ++seniorFemales;
```

  - **Combined condition is `true`**
    - **if and only if both simple conditions are `true`**
  - **Combined condition is `false`**
    - **if either or both of the simple conditions are `false`**

expression1	expression2	expression1 && expression2
false	false	False
false	true	False
true	false	False
true	true	True

Fig. 2c.14 | && (conditional AND) operator truth table.

2c.8 Logical Operators (Cont.)

- **Conditional OR (||) Operator**
  - **Consider the following `if` statement**

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 )
   System.out.println( “Student grade is A” );
```

  - **Combined condition is `true`**
    - **if either or both of the simple condition are `true`**
  - **Combined condition is `false`**
    - **if both of the simple conditions are `false`**

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

Fig. 2c.15 **|| (conditional OR) operator truth table.**

2c.8 Logical Operators (Cont.)

- **Short-Circuit Evaluation of Complex Conditions**
  - **Parts of an expression containing && or || operators are evaluated only until it is known whether the condition is true or false**
  - **E.g., `( gender == FEMALE ) && ( age >= 65 )`**
    - **Stops immediately if gender is not equal to FEMALE**

Common Programming Error 2c.8

In expressions using operator &&, a condition—we will 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.

2c.8 Logical Operators (Cont.)

- **Boolean Logical AND (&) Operator**
  - **Works identically to &&**
  - **Except & always evaluate both operands**
- **Boolean Logical OR (|) Operator**
  - **Works identidally to ||**
  - **Except | always evaluate both operands**

Error-Prevention Tip 2c.4

For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they can make it harder to understand code and can lead to subtle logic errors.

2c.8 Logical Operators (Cont.)

- **Boolean Logical Exclusive OR (^)**
  - **One of its operands is `true` and the other is `false`**
    - **Evaluates to `true`**
  - **Both operands are `true` or both are `false`**
    - **Evaluates to `false`**
- **Logical Negation (!) Operator**
  - **Unary operator**

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 2c.16 | ^ (boolean logical exclusive OR) operator truth table.

expression	!expression
false	true
true	false

Fig. 2c.17 |! (logical negation, or logical NOT) operator truth table.

Outline

LogicalOperators.java

(1 of 3)

Lines 9-13

Lines 16-20

```
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
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
```

Conditional AND truth table

Conditional OR truth table

Boolean logical AND truth table

Outline

```
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
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
```

Boolean logical inclusive OR truth table

Boolean logical exclusive OR truth table

Logical negation truth table

(2 of 3)

Lines 30-35

Lines 46-47

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

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

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

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

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

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

Outline

LogicalOperators.java

(3 of 3)

Program output

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. 2c.19 | Precedence/associativity of the operators discussed so far.

2c.9 Structured Programming Summary

- **Sequence structure**
  - **"built-in" to Java**
- **Selection structure**
  - **`if`, `if…else` and `switch`**
- **Repetition structure**
  - **`while`, `do…while` and `for`**

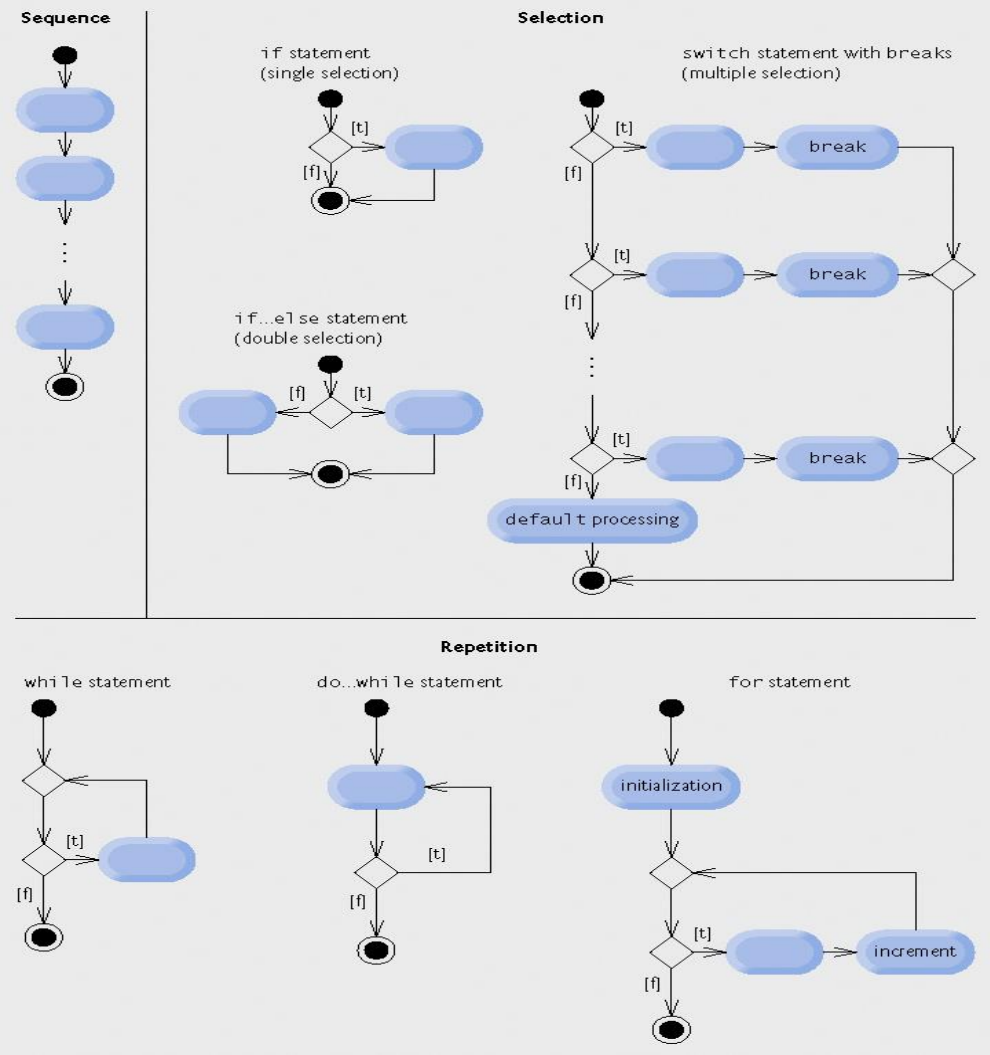


Fig. 2c.20 | Java's single-entry/single-exit sequence, selection and repetition statements.

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.
3	Any action state can be replaced by any control statement (sequence of action states, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).
4	Rules 2 and 3 can be applied as often as you like and in any order.

Fig. 2c.21 | Rules for forming structured programs.

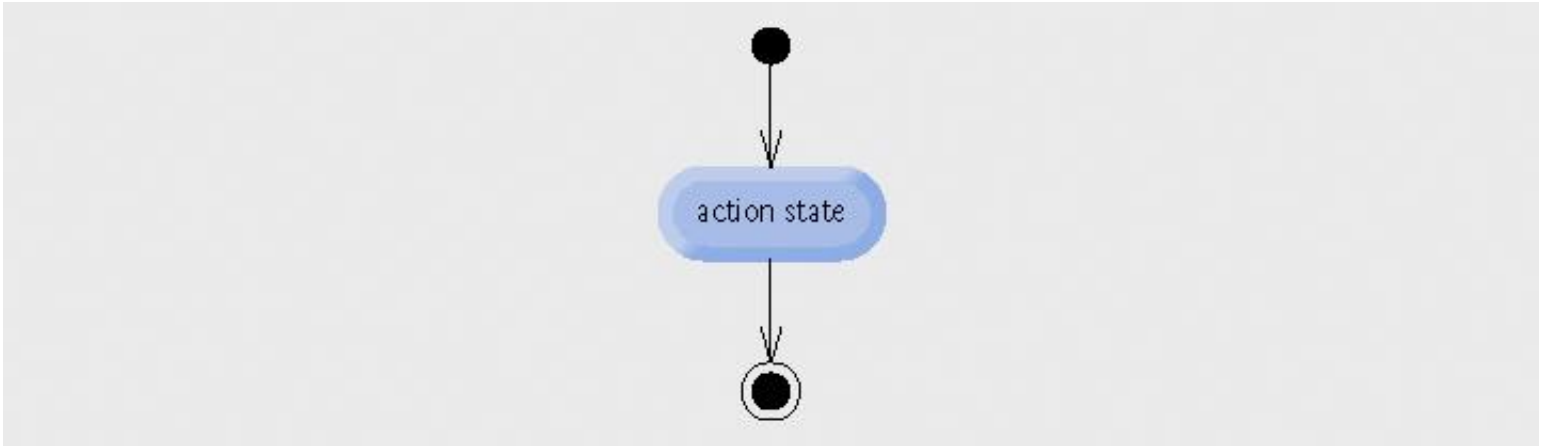


Fig. 2c.22 | Simplest activity diagram.

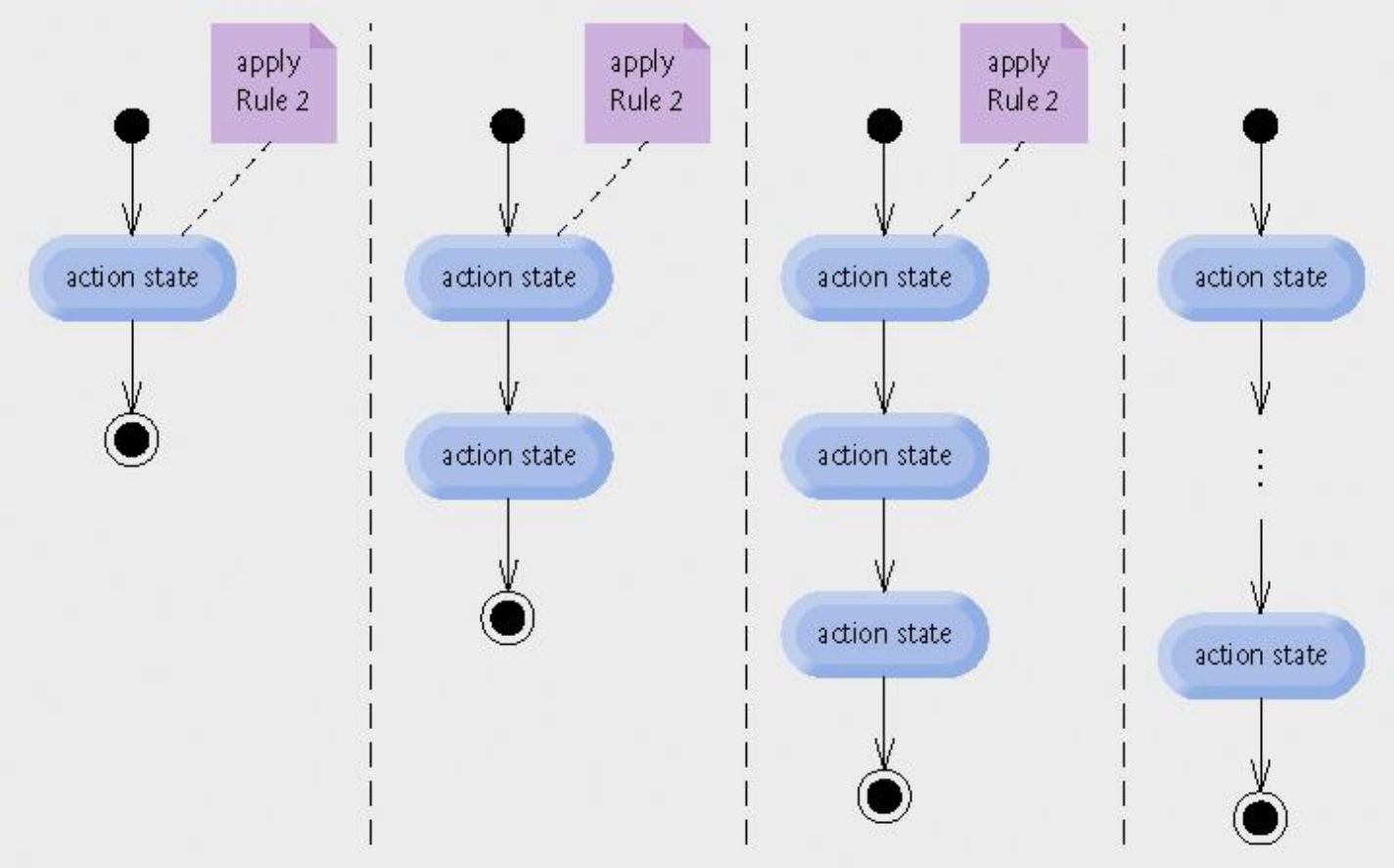


Fig. 2c.23 | Repeatedly applying the stacking rule (rule 2) of Fig. 2c.21 to the simplest activity diagram.

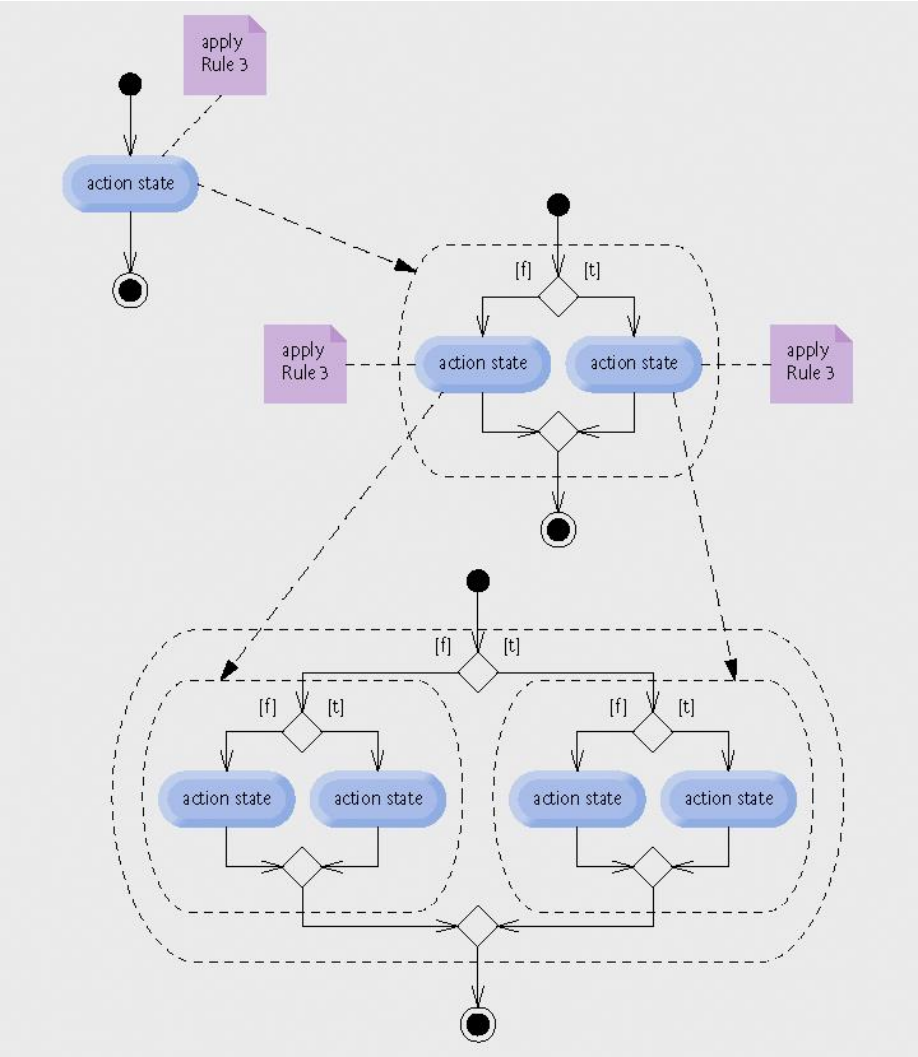


Fig. 2c.24 | Repeatedly applying the nesting rule (rule 3) of Fig. 2c.21 to the simplest activity diagram.

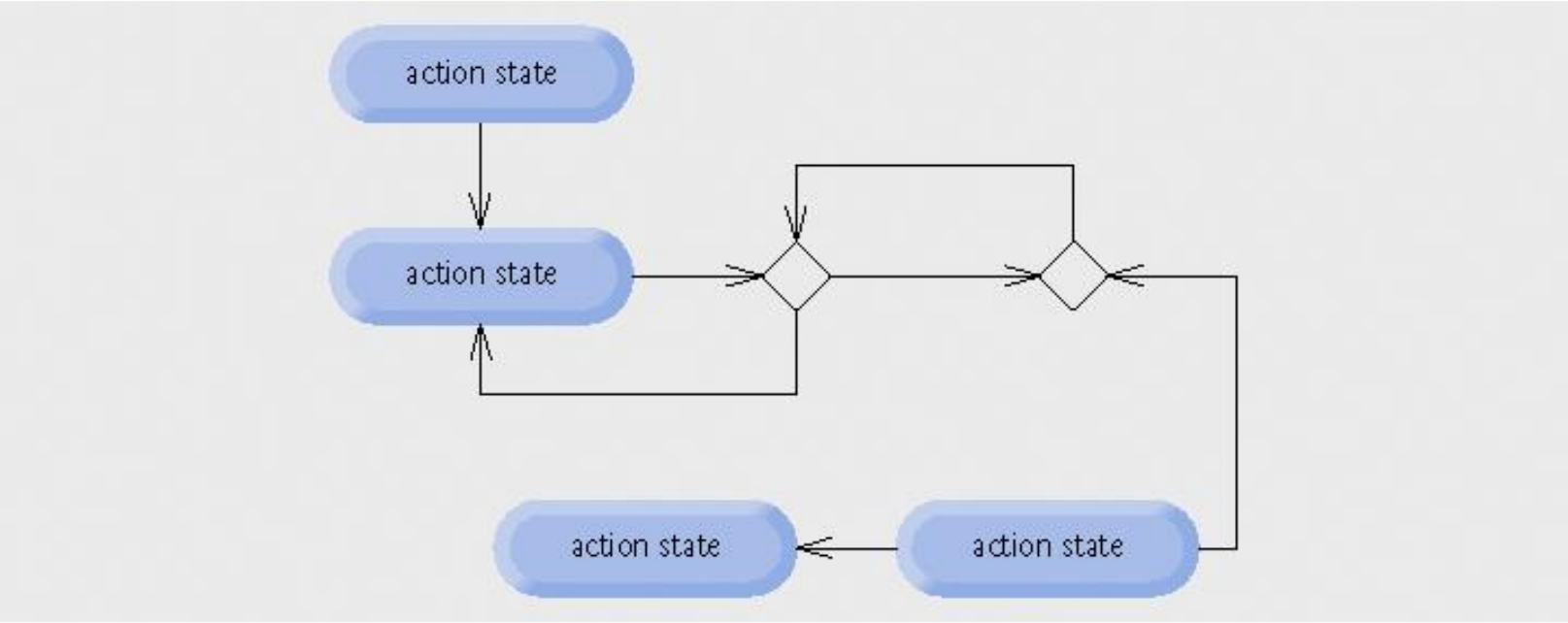


Fig. 2c.25 | "Unstructured" activity diagram.

2c.10 (Optional) GUI and Graphics Case Study: Drawing Rectangles and Ovals

- **Draw rectangles**
  - **Method `drawRect` of `Graphics`**
- **Draw ovals**
  - **Method `drawOval` of `Graphics`**

Outline

Shapes.java

(1 of 2)

```
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
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
```

Draw rectangles

Draw ovals

Outline

Shapes.java

(2 of 2)

Lines 27-28

Lines 31-32

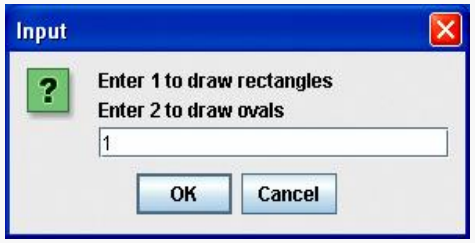
Outline

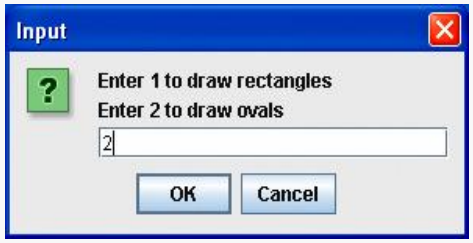
ShapesTest.java

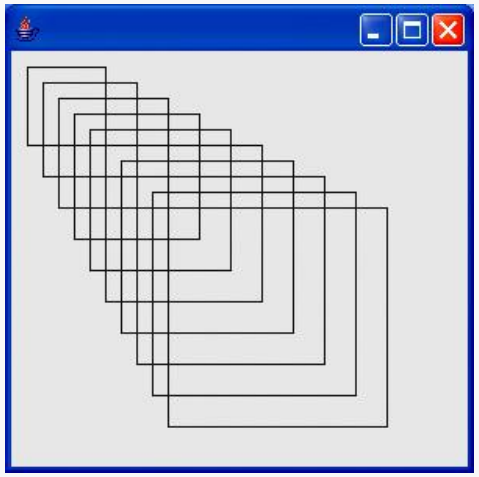
(1 of 2)

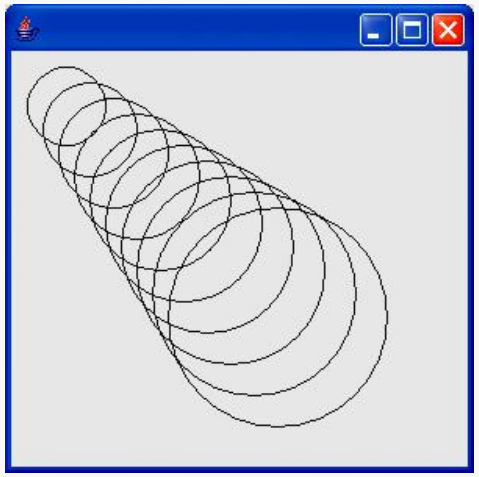
```
1  // Fig. 5.27: ShapesTest.java
2  // Test application that displays class Shapes.
3  import javax.swing.JFrame;
4  import javax.swing.JOptionPane;
5
6  public class ShapesTest
7  {
8     public static void main( String args[] )
9     {
10       // obtain user's choice
11       String input = JOptionPane.showInputDialog(
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
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
```

Outline









`ShapesTest.java`

(2 of 2)

Program output

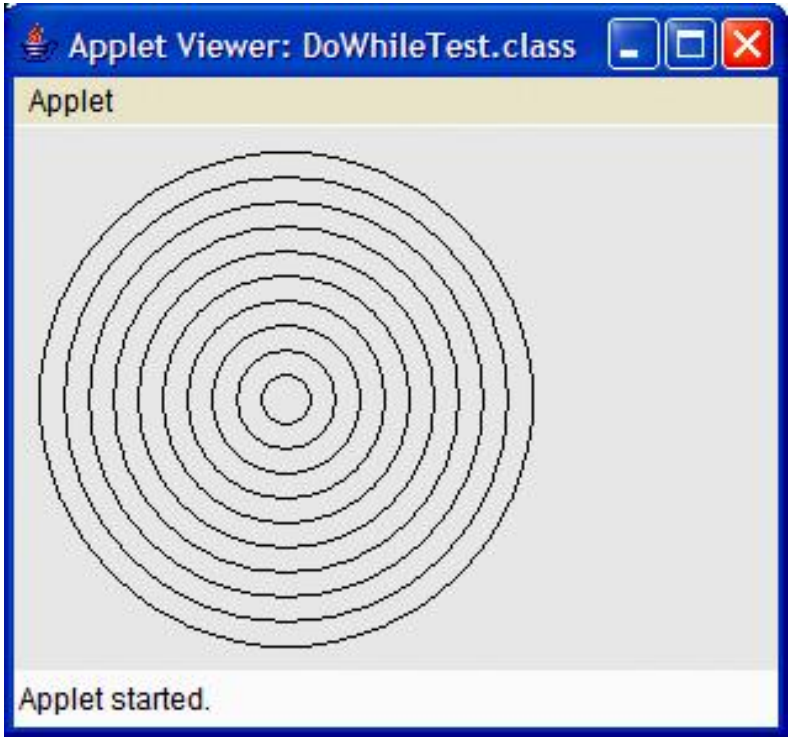


Fig. 2c.28 | Drawing concentric circles.

2c.11 (Optional) Software Engineering Case Study: Identifying Object's State and Activities

- **State Machine Diagrams**
  - **Commonly called state diagram**
  - **Model several states of an object**
  - **Show under what circumstances the object changes state**
  - **Focus on system behavior**
  - **UML representation**
    - **State**
      - **Rounded rectangle**
    - **Initial state**
      - **Solid circle**
    - **Transitions**
      - **Arrows with stick arrowheads**

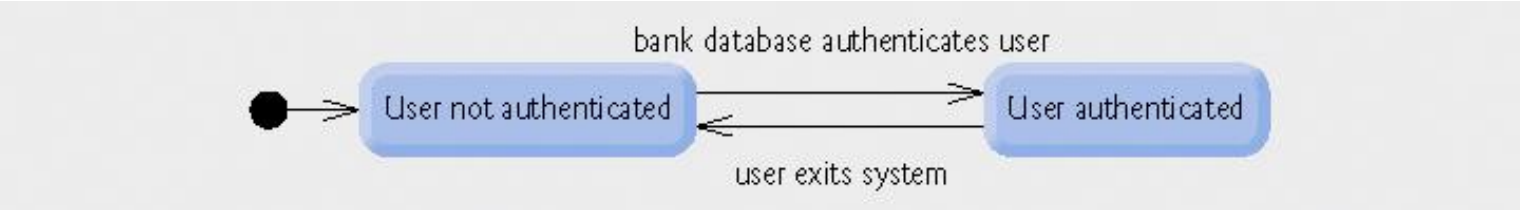


Fig. 2c.29 | State diagram for the ATM object.

Software Engineering Observation 2c.5

Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only key states and state transitions.

2c.11 (Optional) Software Engineering Case Study (Cont.)

- **Activity Diagrams**
  - **Focus on system behavior**
  - **Model an object's workflow during program execution**
  - **Model the actions the object will perform and in what order**
  - **UML representation**
    - **Action state (rectangle with its left and right sides replaced by arcs curving outwards)**
    - **Action order (arrow with a stick arrowhead)**
    - **Initial state (solid circle)**
    - **Final state (solid circle enclosed in an open circle)**

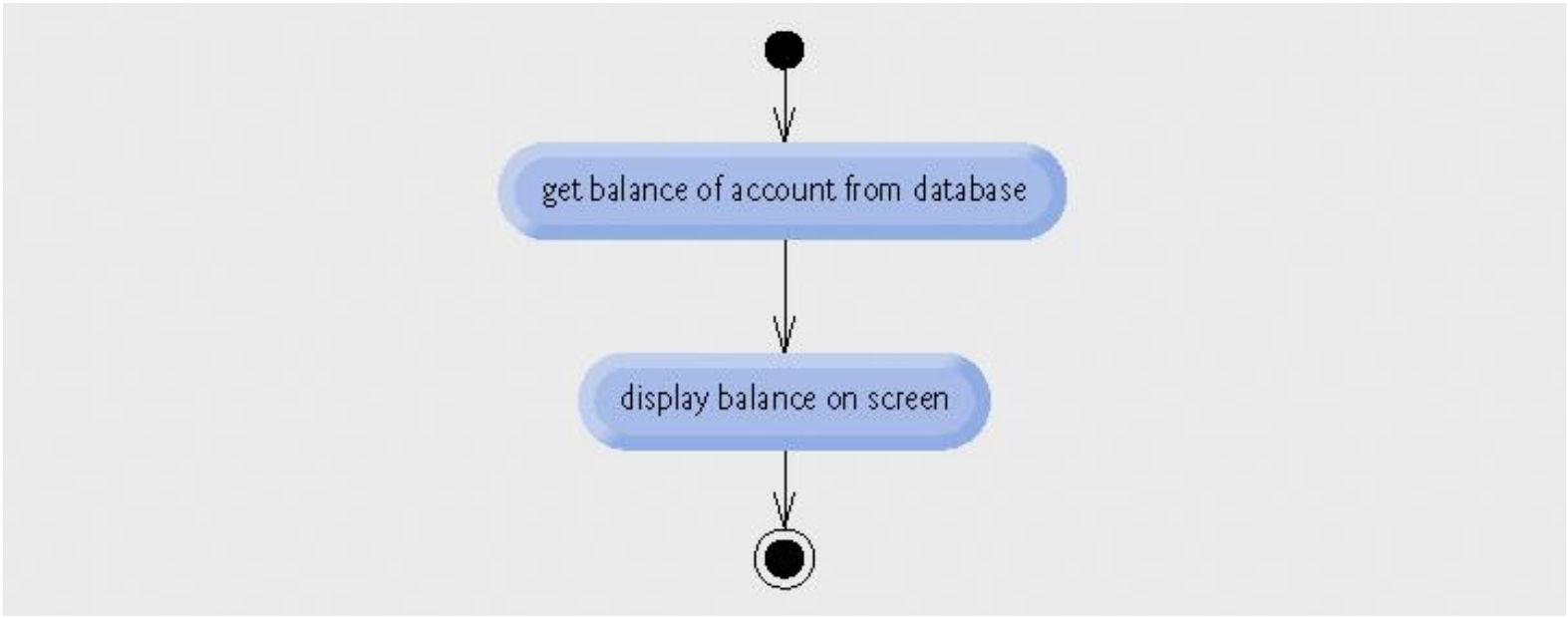


Fig. 2c.30 | Activity diagram for a BalanceInquiry object.

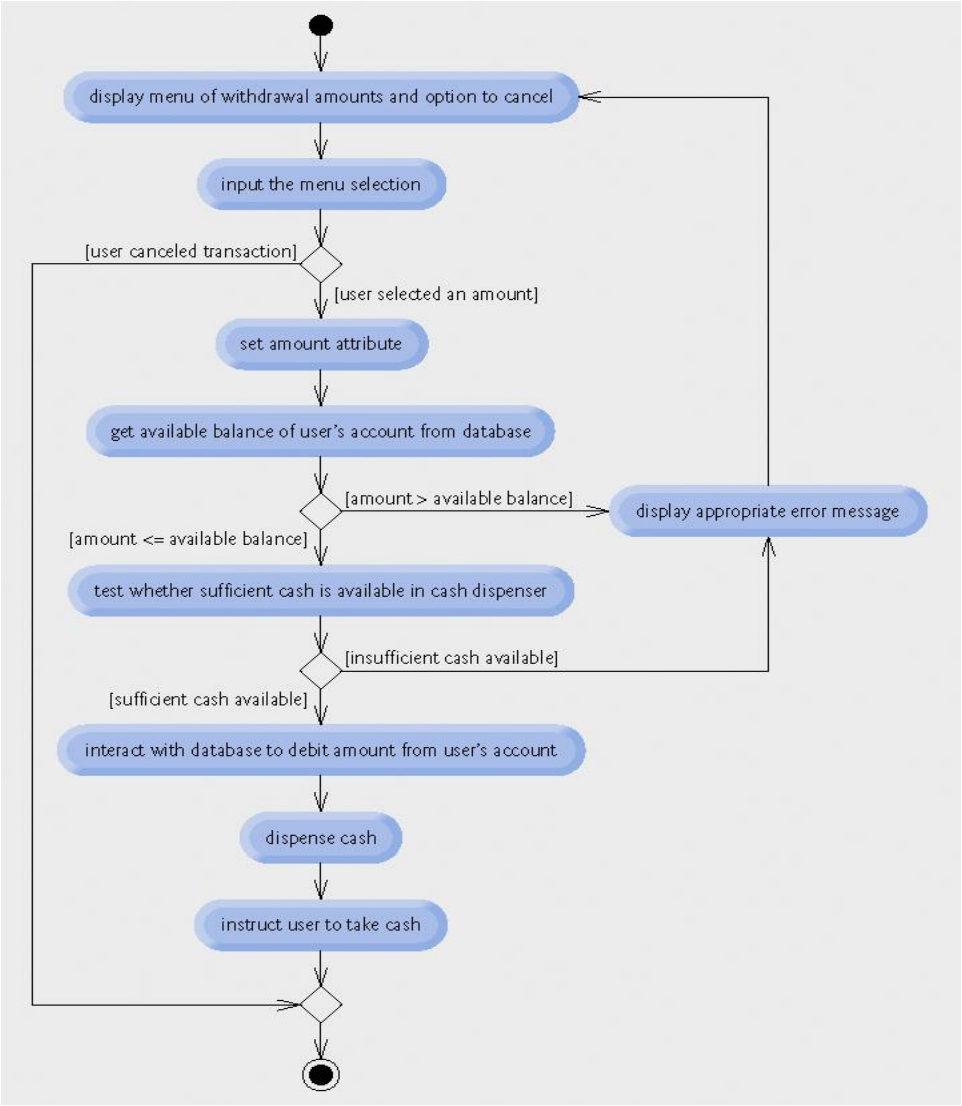


Fig. 2c.31 | Activity diagram for a withdrawal transaction.

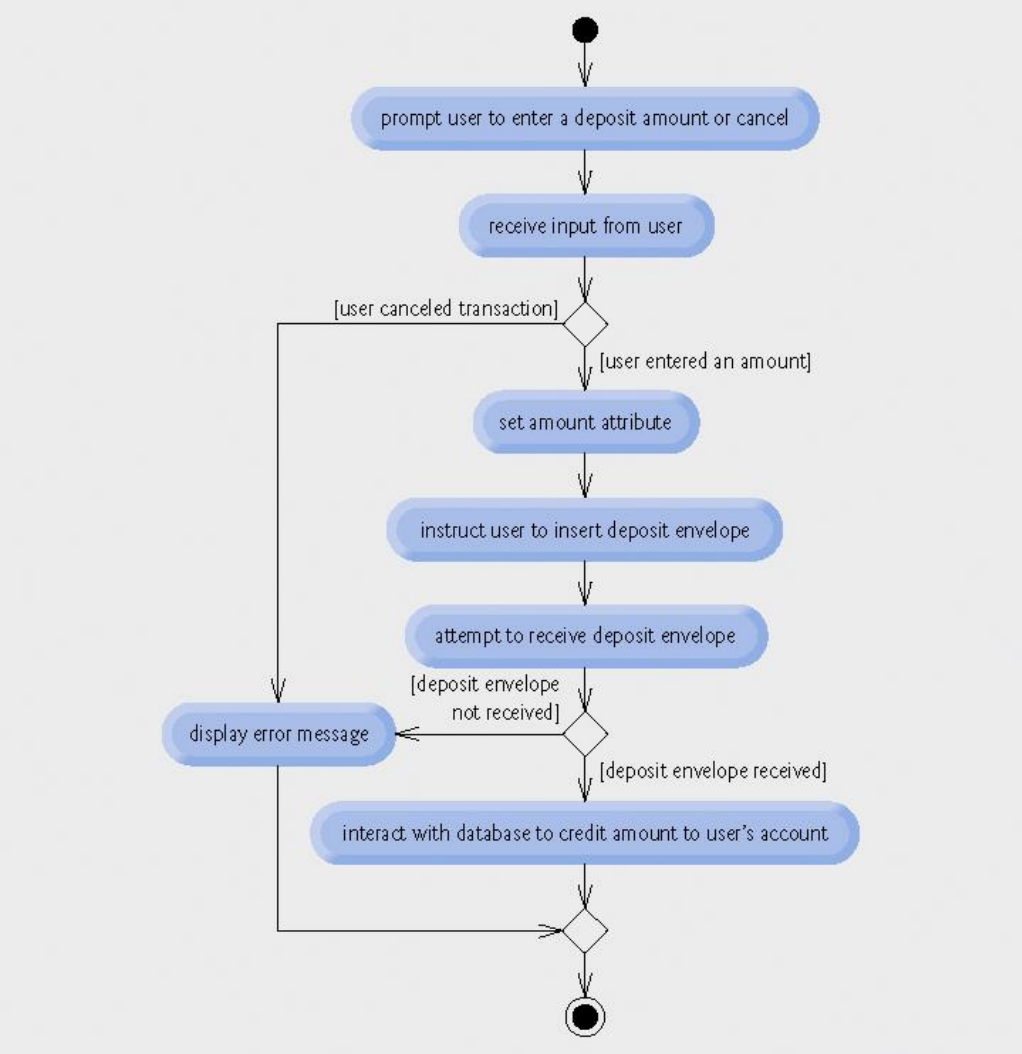


Fig. 2c.32 | Activity diagram for a deposit transaction.