Chapter 5 – Control Structures: Part 2

Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - Name of control variable (loop counter)
  - Initial value of control variable
  - Increment/decrement of control variable through each loop
  - Condition that tests for control variable's final value

1

The `for` Repetition Structure (cont.)

```
for ( expression1; expression2; expression3 )
   statement;
```

can usually be rewritten as:

```
expression1;
while ( expression2 ) {
   statement;
   expression3;
}
```

2

Examples Using the `for` Structure

- Varying control variable in **`for`** structure
  - 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 decrements of **`-1`**
    - **`for ( int i = 100; i >= 1; i-- )`**
  - Vary control variable from **`7`** to **`77`** in steps of **`7`**
    - **`for ( int i = 7; i <= 77; i += 7 )`**

3

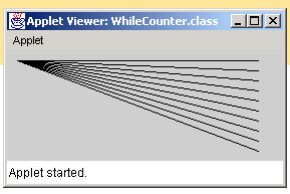
```
1  // Fig. 5.1: WhileCounter.java
2  // Counter-controlled repetition
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.JApplet;
9
10 public class WhileCounter extends JApplet {
11
12    // draw lines on applet's background
13    public void paint( Graphics g )
14    {
15       // call inherited version of method paint
16       super.paint( g );
17
18       int counter = 1;             // initialization
19
20       while ( counter <= 10 ) {    // repetition condition
21          g.drawLine( 10, 10, 250, counter * 10 );
22          ++counter;                // increment
23
24       }  // end while structure
25
26    }  // end method paint
27
28 }  // end class WhileCounter
```

`WhileCounter.java`

Line 18

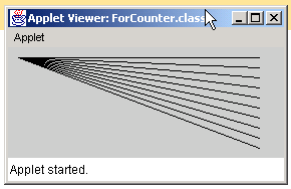
Line 20

Line 22



4

```
1  // Fig. 5.2: ForCounter.java
2  // Counter-controlled repetition with the for structure
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.JApplet;
9
10 public class ForCounter extends JApplet {
11
12    // draw lines on applet's background
13    public void paint( Graphics g )
14    {
15       // call inherited version of method paint
16       super.paint( g );
17
18       // Initialization, repetition condition and incrementing
19       // are all included in the for structure header.
20       for ( int counter = 1; counter <= 10; counter++ )
21          g.drawLine( 10, 10, 250, counter * 10 );
22
23    }  // end method paint
24
25 }  // end class ForCounter
```



ForCounter.java

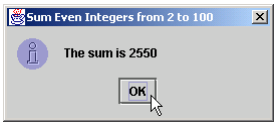
Line 20
`int counter = 1;`

Line 20
`counter <= 10;`

Line 20
`counter++;`

5

```
1  // Fig. 5.5: Sum.java
2  // Counter-controlled repetition with the for structure
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class Sum {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       int sum = 0;
13
14       // sum even integers from 2 through 100
15       for ( int number = 2; number <= 100; number += 2 )
16          sum += number;
17
18       // display results
19       JOptionPane.showMessageDialog( null, "The sum is " + sum,
20          "Sum Even Integers from 2 to 100",
21          JOptionPane.INFORMATION_MESSAGE );
22
23       System.exit( 0 );   // terminate the application
24
25    }  // end method main
26
27 }  // end class Sum
```



Sum.java

Line 15

6

```
1  // Fig. 5.6: Interest.java
2  // Calculating compound interest
3
4  // Java core packages
5  import java.text.NumberFormat;
6  import java.util.Locale;
7
8  // Java extension packages
9  import javax.swing.JOptionPane;
10 import javax.swing.JTextArea;
11
12 public class Interest {
13
14    // main method begins execution of Java application
15    public static void main( String args[] )
16    {
17       double amount, principal = 1000.0, rate = 0.05;
18
19       // create DecimalFormat to format floating-point numbers
20       // with two digits to the right of the decimal point
21       NumberFormat moneyFormat =
22          NumberFormat.getCurrencyInstance( Locale.US );
23
24       // create JTextArea to display output
25       JTextArea outputTextArea = new JTextArea();
26
27       // set first line of text in outputTextArea
28       outputTextArea.setText( "Year\tAmount on deposit\n" );
29
30       // calculate amount on deposit for each of ten years
31       for ( int year = 1; year <= 10; year++ ) {
32
33          // calculate new amount for specified year
34          amount = principal * Math.pow( 1.0 + rate, year );
35
```

Interest.java

Line 17

Line 21

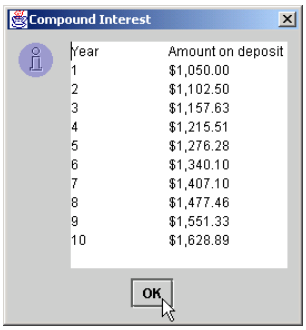
Line 22

Lines 31-34

7

```
36          // append one line of text to outputTextArea
37          outputTextArea.append( year + "\t" +
38             moneyFormat.format( amount ) + "\n" );
39
40       }  // end for structure
41
42       // display results
43       JOptionPane.showMessageDialog( null, outputTextArea,
44          "Compound Interest", JOptionPane.INFORMATION_MESSAGE );
45
46       System.exit( 0 );  // terminate the application
47
48    }  // end method main
49
50 }  // end class Interest
```

Interest.java



8

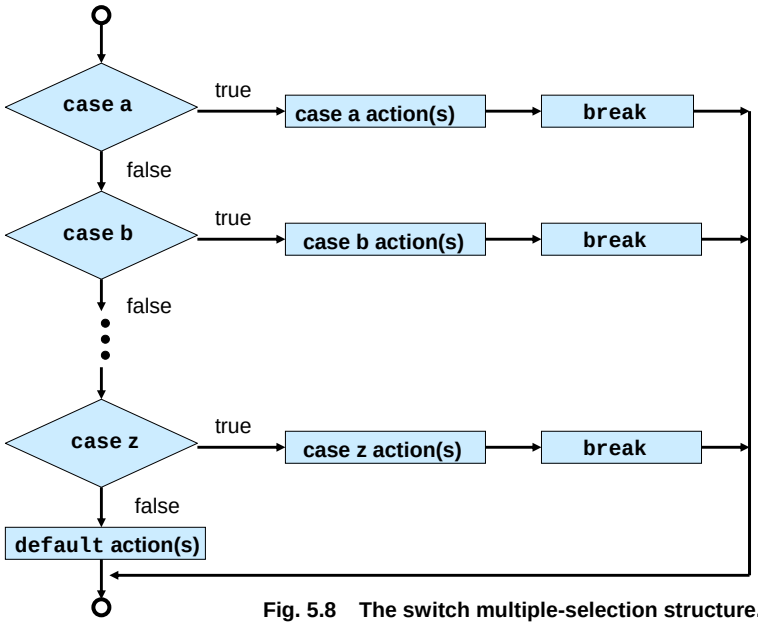


Fig. 5.8 The switch multiple-selection structure.

9

```
1  // Fig. 5.7: SwitchTest.java
2  // Drawing lines, rectangles or ovals based on user input.
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class SwitchTest extends JApplet {
11    int choice;   // user's choice of which shape to draw
12
13    // initialize applet by obtaining user's choice
14    public void init()
15    {
16       String input;  // user's input
17
18       // obtain user's choice
19       input = JOptionPane.showInputDialog(
20                  "Enter 1 to draw lines\n" +
21                  "Enter 2 to draw rectangles\n" +
22                  "Enter 3 to draw ovals\n" );
23
24       // convert user's input to an int
25       choice = Integer.parseInt( input );
26    }
27
28    // draw shapes on applet's background
29    public void paint( Graphics g )
30    {
31       // call inherited version of method paint
32       super.paint( g );
33
```

`SwitchTest.java`

Lines 19-25:
Getting user's input

10

```
34       // loop 10 times, counting from 0 through 9
35       for ( int i = 0; i < 10; i++ ) {
36
37          // determine shape to draw based on user's choice
38          switch ( choice ) {
39
40             case 1:
41                g.drawLine( 10, 10, 250, 10 + i * 10 );
42                break;  // done processing case
43
44             case 2:
45                g.drawRect( 10 + i * 10, 10 + i * 10,
46                   50 + i * 10, 50 + i * 10 );
47                break;  // done processing case
48
49             case 3:
50                g.drawOval( 10 + i * 10, 10 + i * 10,
51                   50 + i * 10, 50 + i * 10 );
52                break;  // done processing case
53
54             default:
55                g.drawString( "Invalid value entered",
56                   10, 20 + i * 15 );
57
58          }  // end switch structure
59
60       }  // end for structure
61
62    }  // end paint method
63
64 }  // end class SwitchTest
```

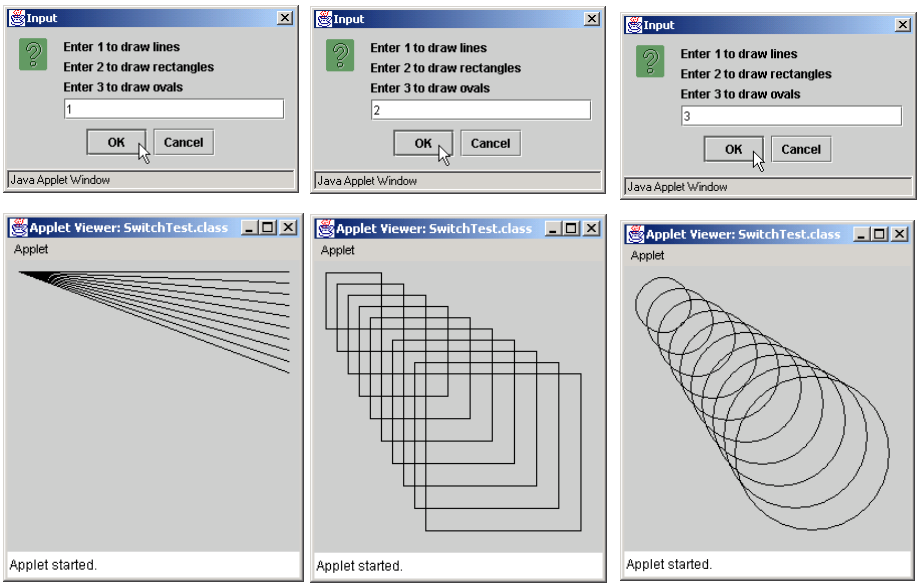
`SwitchTest.java`

Line 38:
controlling expression

Line 38:
`switch` structure

Line 54

11



12

The do/while Repetition Structure

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

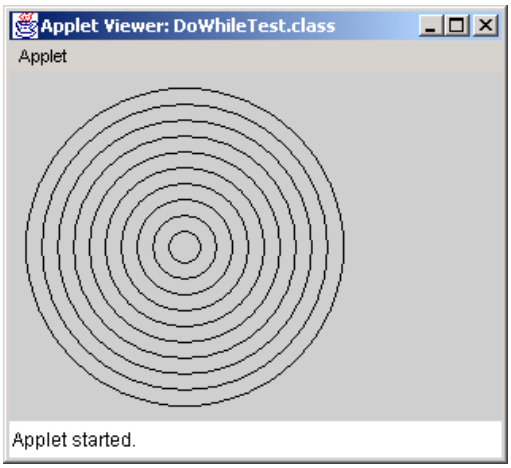
13

```
1  // Fig. 5.9: DoWhileTest.java
2  // Using the do/while repetition structure.
3
4  // Java core packages
5  import java.awt.Graphics;
6
7  // Java extension packages
8  import javax.swing.JApplet;
9
10 public class DoWhileTest extends JApplet {
11
12    // draw lines on applet's background
13    public void paint( Graphics g )
14    {
15       // call inherited version of method paint
16       super.paint( g );
17
18       int counter = 1;
19
20       do {
21          g.drawOval( 110 - counter * 10, 110 - counter * 10,
22             counter * 20, counter * 20 );
23          ++counter;
24       } while ( counter <= 10 );  // end do/while structure
25
26    }  // end method paint
27
28 }  // end class DoWhileTest
```

DoWhileTest.java

Lines 20-24

14



DoWhileTest.java

15

Statements `break` and `continue`

- **`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

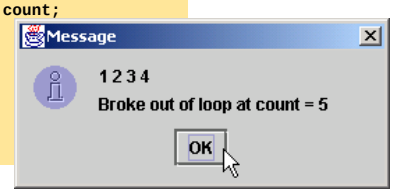
16

```
1  // Fig. 5.11: BreakTest.java
2  // Using the break statement in a for structure
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class BreakTest {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       String output = "";
13       int count;
14
15       // loop 10 times
16       for ( count = 1; count <= 10; count++ ) {
17
18          // if count is 5, terminate loop
19          if ( count == 5 )
20             break;  // break loop only if count == 5
21
22          output += count + " ";
23
24       }  // end for structure
25
26       output += "\nBroke out of loop at count = " + count;
27       JOptionPane.showMessageDialog( null, output );
28
29       System.exit( 0 );  // terminate application
30
31    }  // end method main
32
33 }  // end class BreakTest
```

BreakTest.java

Line 16

Lines 19-20




```
1  // Fig. 5.12: ContinueTest.java
2  // Using the continue statement in a for structure
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class ContinueTest {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       String output = "";
13
14       // loop 10 times
15       for ( int count = 1; count <= 10; count++ ) {
16
17          // if count is 5, continue with next iteration of loop
18          if ( count == 5 )
19             continue;  // skip remaining code in loop
20                        // only if count == 5
21
22          output += count + " ";
23
24       }  // end for structure
25
26       output += "\nUsed continue to skip printing
27       JOptionPane.showMessageDialog( null, output
28
29       System.exit( 0 );  // terminate application
30
31    } // end method main
32
33 }  // end class ContinueTest
```

ContinueTest.java

Line 15

Lines 18-19

18

Labeled `break` and `continue` Statements

- Labeled block
  - Set of statements enclosed by **`{}`**
  - Preceded by a label
- Labeled **`break`** statement
  - Exit from nested control structures
  - Proceeds to end of specified labeled block
- Labeled **`continue`** statement**(should have a loop)**
  - Skips remaining statements in nested-loop body
  - Proceeds to beginning of specified labeled block

19

```
1  // Fig. 5.13: BreakLabelTest.java
2  // Using the break statement with a label
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class BreakLabelTest {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       String output = "";
13
14       stop: {   // labeled block
15
16          // count 10 rows
17          for ( int row = 1; row <= 10; row++ ) {
18
19             // count 5 columns
20             for ( int column = 1; column <= 5 ; column++ ) {
21
22                // if row is 5, jump to end of "stop" block
23                if ( row == 5 )
24                   break stop; // jump to end of stop block
25
26                output += "*  ";
27
28             }  // end inner for structure
29
30             output += "\n";
31
32          }  // end outer for structure
33
34          // the following line is skipped
35          output += "\nLoops terminated normally";
```

`BreakLabelTest.java`

Line 14

Line 17

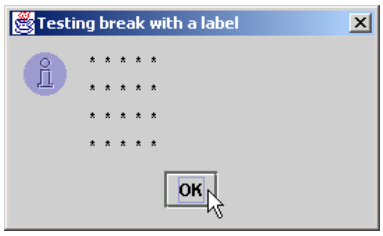
Line 20

Lines 23-24

20

```
36
37       }  // end labeled block
38
39       JOptionPane.showMessageDialog(
40          null, output,"Testing break with a label",
41          JOptionPane.INFORMATION_MESSAGE );
42
43       System.exit( 0 );  // terminate application
44
45    }  // end method main
46
47 }  // end class BreakLabelTest
```

BreakLabelTest.java



21

```
1  // Fig. 5.14: ContinueLabelTest.java
2  // Using the continue statement with a label
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class ContinueLabelTest {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       String output = "";
13
14       nextRow:    // target label of continue statement
15
16          // count 5 rows
17          for ( int row = 1; row <= 5; row++ ) {
18             output += "\n";
19
20             // count 10 columns per row
21             for ( int column = 1; column <= 10; column++ ) {
22
23                // if column greater than row, start next row
24                if ( column > row )
25                   continue nextRow; // next iteration of
26                                     // labeled loop
27
28                output += "*  ";
29
30             }  // end inner for structure
31
32          }  // end outer for structure
33
```

ContinueLabelTest.java

Line 14

Line 17

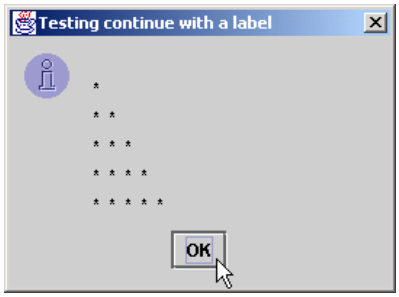
Line 21

Lines 24-25

22

```
34         JOptionPane.showMessageDialog(
35            null, output,"Testing continue with a label",
36            JOptionPane.INFORMATION_MESSAGE );
37
38         System.exit( 0 );  // terminate application
39
40      }  // end method main
41
42   }  // end class ContinueLabelTest
```

ContinueLabelTest.java



23

Logical Operators

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

24

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

Fig. 5.15 Truth table for the logical exclusive OR (^) operator.

25

```
1  // Fig. 5.19: LogicalOperators.java
2  // Demonstrating the logical operators
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class LogicalOperators {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       // create JTextArea to display results
13       JTextArea outputArea = new JTextArea( 17, 20 );
14
15       // attach JTextArea to a JScrollPane so user can
16       // scroll through results
17       JScrollPane scroller = new JScrollPane( outputArea );
18
19       String output;
20
21       // create truth table for && operator
22       output = "Logical AND (&&)" +
23          "\nfalse && false: " + ( false && false ) +
24          "\nfalse && true: " + ( false && true ) +
25          "\ntrue && false: " + ( true && false ) +
26          "\ntrue && true: " + ( true && true );
27
28       // create truth table for || operator
29       output += "\n\nLogical OR (||)" +
30          "\nfalse || false: " + ( false || false ) +
31          "\nfalse || true: " + ( false || true ) +
32          "\ntrue || false: " + ( true || false ) +
33          "\ntrue || true: " + ( true || true );
34
```

LogicalOperators.java

Lines 22-26

Lines 29-33

26

```
35        // create truth table for & operator
36        output += "\n\nBoolean logical AND (&)" +
37           "\nfalse & false: " + ( false & false ) +
38           "\nfalse & true: " + ( false & true ) +
39           "\ntrue & false: " + ( true & false ) +
40           "\ntrue & true: " + ( true & true );
41
42        // create truth table for | operator
43        output += "\n\nBoolean logical inclusive OR (|)" +
44           "\nfalse | false: " + ( false | false ) +
45           "\nfalse | true: " + ( false | true ) +
46           "\ntrue | false: " + ( true | false ) +
47           "\ntrue | true: " + ( true | true );
48
49        // create truth table for ^ operator
50        output += "\n\nBoolean logical exclusive OR (^)" +
51           "\nfalse ^ false: " + ( false ^ false ) +
52           "\nfalse ^ true: " + ( false ^ true ) +
53           "\ntrue ^ false: " + ( true ^ false ) +
54           "\ntrue ^ true: " + ( true ^ true );
55
56        // create truth table for ! operator
57        output += "\n\nLogical NOT (!)" +
58           "\n!false: " + ( !false ) +
59           "\n!true: " + ( !true );
60
61        outputArea.setText( output );  // place results in JTextArea
62
63        JOptionPane.showMessageDialog( null, scroller,
64           "Truth Tables", JOptionPane.INFORMATION_MESSAGE );
65
66        System.exit( 0 );  // terminate application
```

LogicalOperators.java

Lines 36-40

Lines 43-47

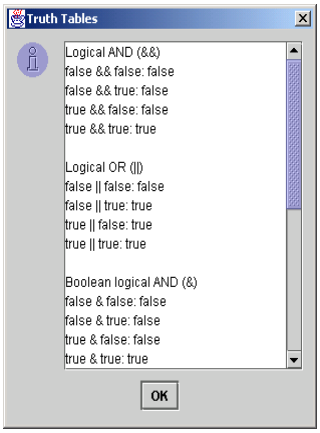
Lines 50-54

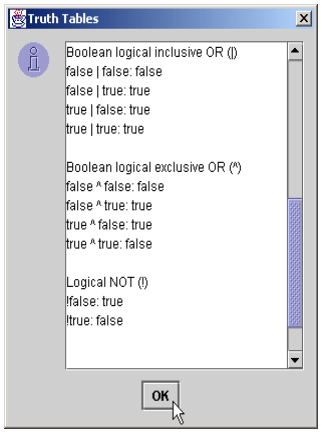
Lines 57-59

27

```
67
68     }  // end method main
69
70  }  // end class LogicalOperators
```

LogicalOperators.java





28

Operators	Associativity	Type
( )	left to right	parentheses
++ --	right to left	unary postfix
++ -- + - ! (*type*)	right to left	unary
* / %	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	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 5.20 Precedence and associativity of the operators discussed so far.

29

Structured Programming Summary

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

30

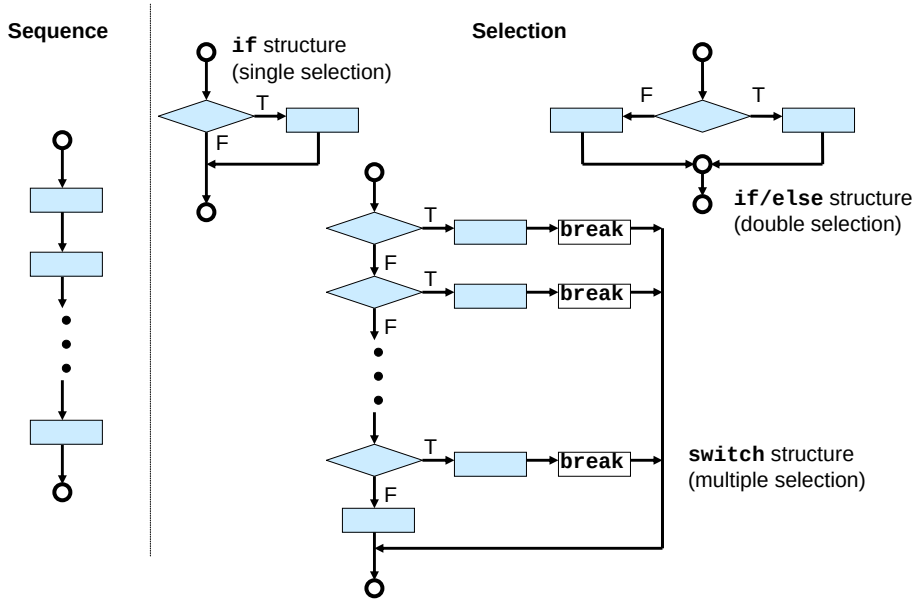


Fig. 5.21 Java's single-entry/single-exit control structures.

31

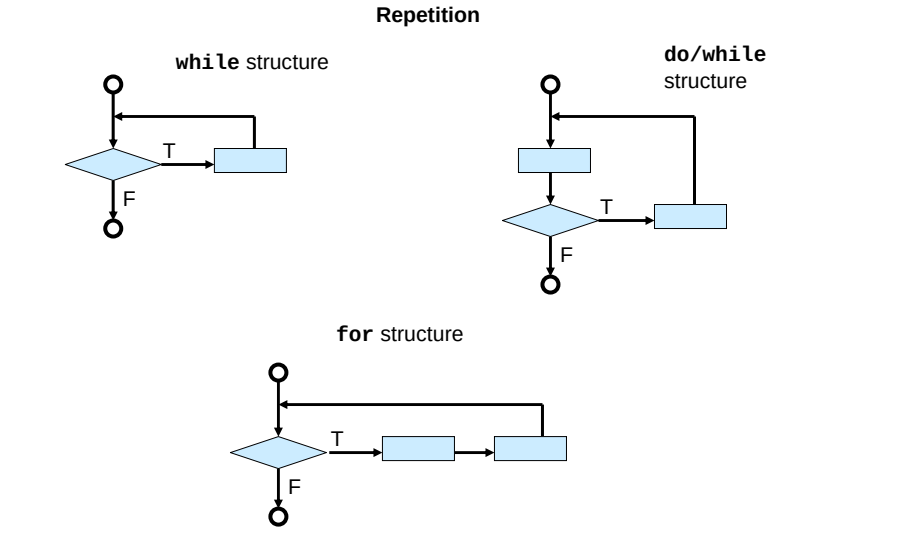


Fig. 5.21 Java's single-entry/single-exit control structures.

32

	Rules for Forming Structured Programs
1)	Begin with the “simplest flowchart” (Fig. 5.23).
2)	Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3)	Any rectangle (action) can be replaced by any control structure (sequence, `if`, `if/else`, `switch`, `while`, `do/while` or `for`).
4)	Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 5.22 Rules for forming structured programs.

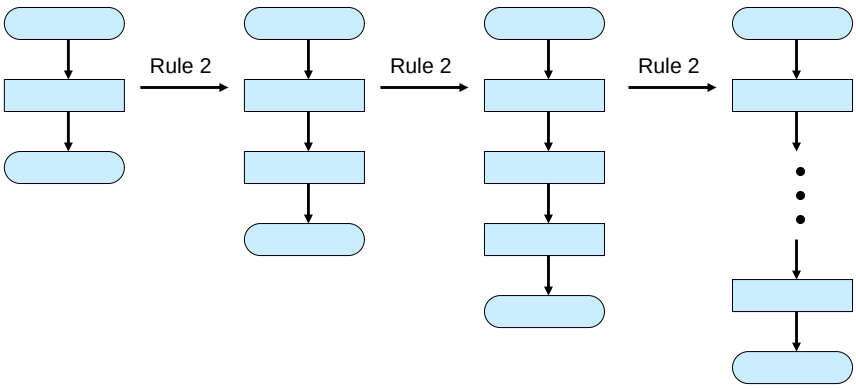


Fig. 5.24 Repeatedly applying Rule 2 of Fig. 5.22 to the simplest flowchart.

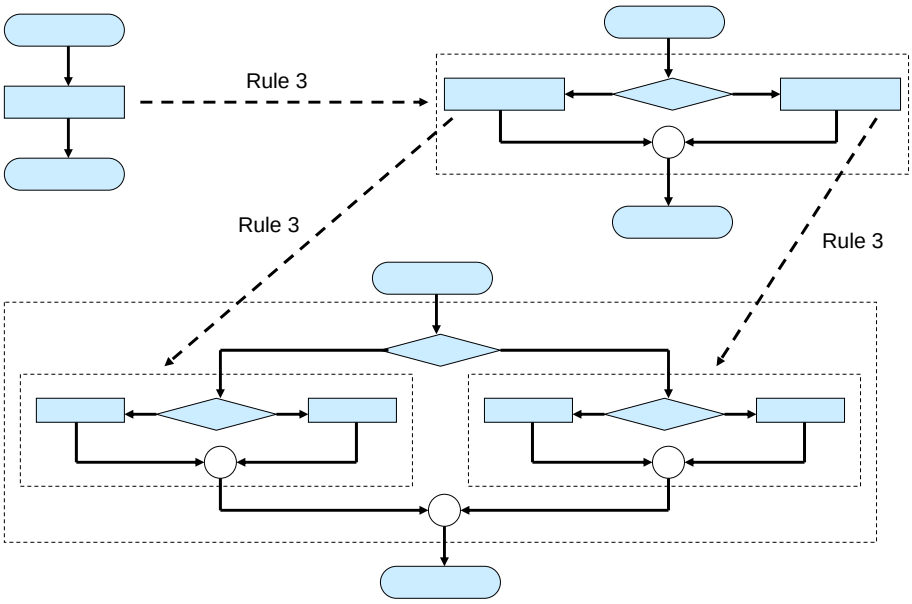


Fig. 5.25 Applying Rule 3 of Fig. 5.22 to the simplest flowchart.

35

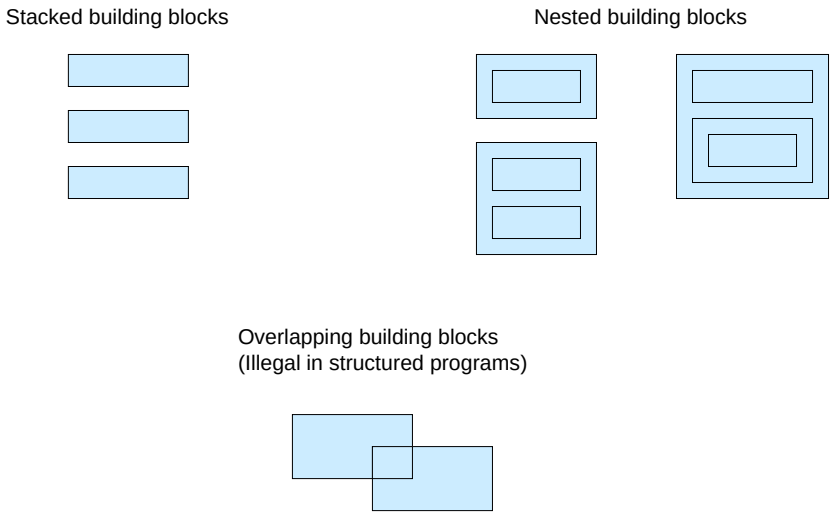


Fig. 5.26 Stacked, nested and overlapped building blocks.

36

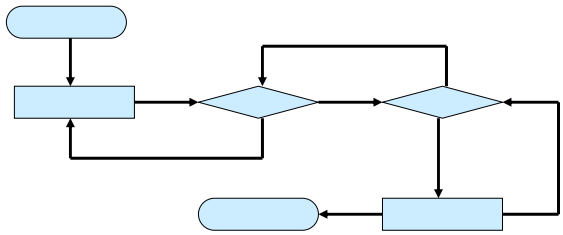


Fig. 5.27 An unstructured flowchart.

(Optional Case Study) Thinking About Objects: Identifying Objects' States and Activities

- State
  - Describes an object's condition at a given time
- Statechart diagram (UML)
  - Express how an object can **change state**
  - Express **under what conditions** an object can change state
  - Diagram notation (Fig. 5.28)
    - **States** are represented by **rounded rectangles**
      - e.g., "**`Not Pressed`**" and "**`Pressed`**"
    - **Solid circle** (with attached arrowhead) designates **initial state**
    - **Arrows** represent **transitions** (state changes)
      - Objects change state in response to messages
        - e.g., "**`buttonPressed`**" and "**`buttonReset`**"

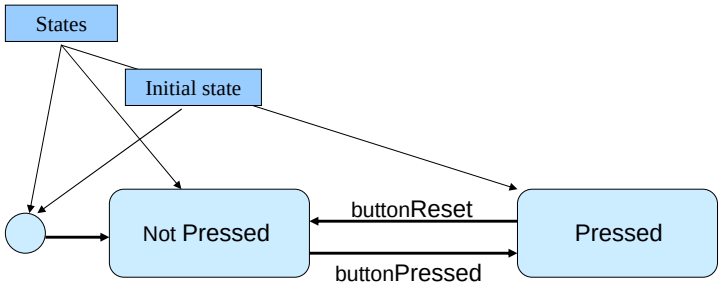


Fig. 5.28 Statechart diagram for `FloorButton` and `ElevatorButton` objects.

39

Thinking About Objects (cont.):

- Activity diagram (UML)
  - Models an object's *workflow* during program execution
  - Models the *actions* that an object will perform
  - Diagram notation (Fig. 5.28)
    - **Activities** are represented by **ovals**
    - **Solid circle** designates initial **activity**
    - **Arrows** represents **transitions** between activities
    - **Small diamond** represents ***branch***
      - Next transition at branch is based on ***guard condition***

40

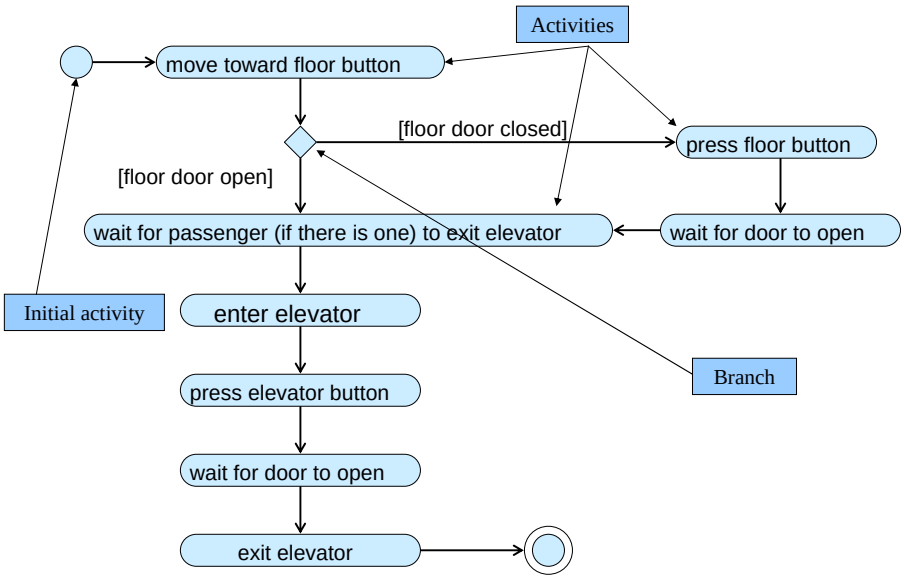


Fig. 5.29 Activity diagram for a `Person` object.

41

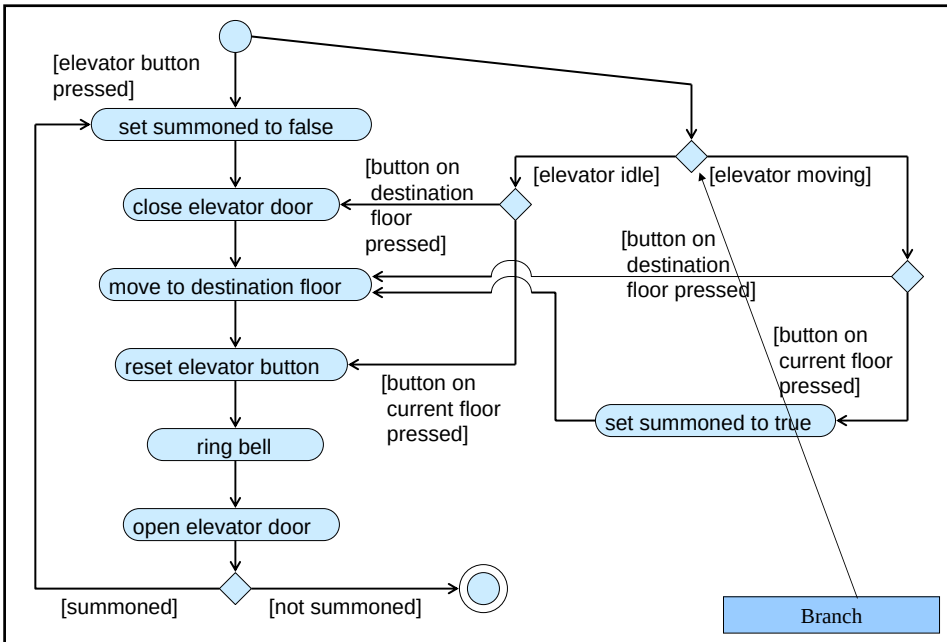


Fig. 5.30 Activity diagram for the `Elevator` object.

42