Chapter 3 Program Control (Main Page)

```
1  // Fig. 3.1: WhileCounter.java
2  // Counter-controlled repetition
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class WhileCounter extends Applet {
7     public void paint( Graphics g )
8     {
9        int counter = 1;             // initialization
10       int yPos = 25;
11
12       while ( counter <= 10 ) {    // repetition condition
13          g.drawString( Integer.toString( counter ),
14                        25, yPos );
15          ++counter;                // increment
16          yPos += 15;
17       }
18    }
19 }
```

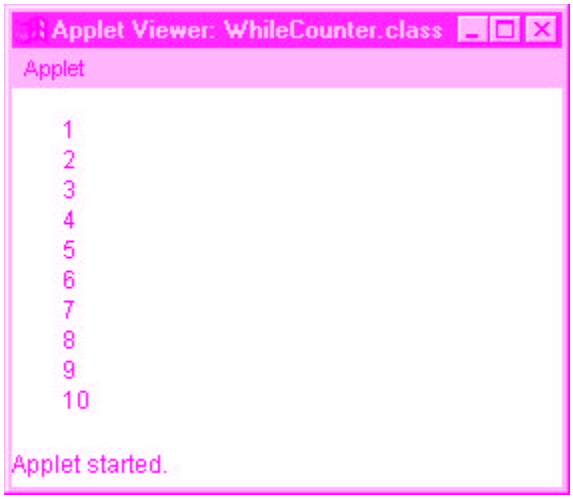


Fig. 3.1 Counter-controlled repetition.

```
1  // Fig. 3.2: ForCounter.java
2  // Counter-controlled repetition with the for structure
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class ForCounter extends Applet {
7     public void paint( Graphics g )
8     {
9        int yPos = 25;
10
11       // Initialization, repetition condition, and incrementing
12       // are all included in the for structure header.
13       for ( int counter = 1; counter <= 10; counter++ ) {
14          g.drawString( Integer.toString( counter ), 25, yPos );
15          yPos += 15;
16       }
17    }
18 }
```

Fig. 3.2 Counter-controlled repetition with the `for` structure.

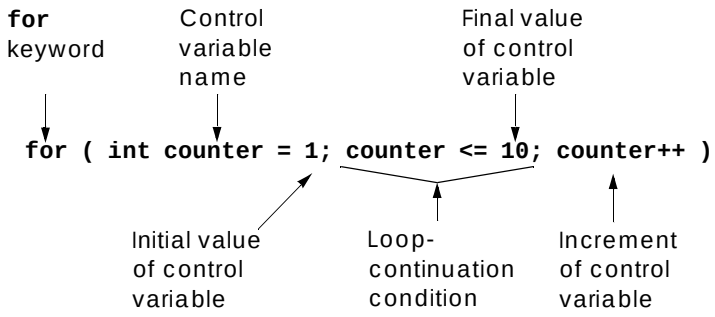


Fig. 3.3 Components of a typical `for` header.

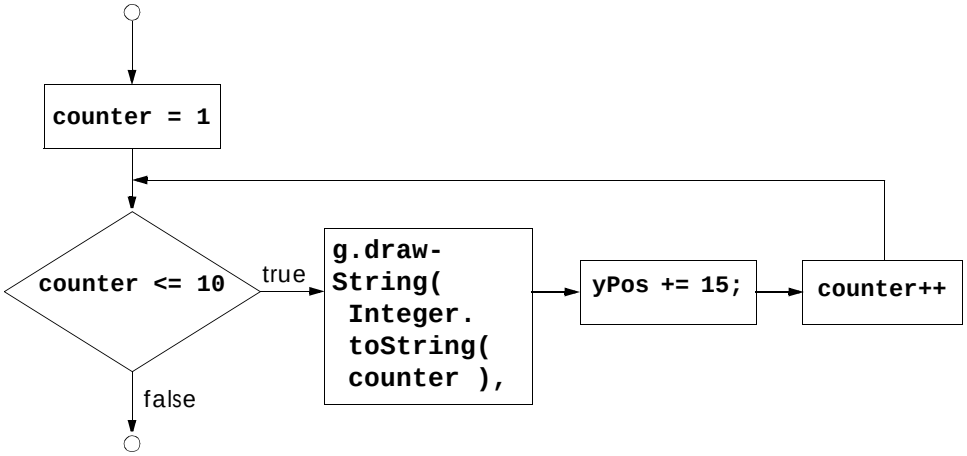
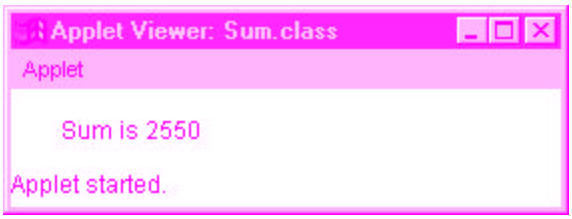


Fig. 3.4 Flowcharting a typical `for` repetition structure.

```
 1  // Fig. 3.5: Sum.java
 2  // Counter-controlled repetition with the for structure
 3  import java.awt.Graphics;
 4  import java.applet.Applet;
 5
 6  public class Sum extends Applet {
 7     public void paint( Graphics g )
 8     {
 9        int sum = 0;
10
11        for ( int number = 2; number <= 100; number += 2 )
12           sum += number;
13
14        g.drawString( "Sum is " + sum, 25, 25 );
15     }
16  }
```

Fig. 3.5 Summation with `for`.



.

```
1  // Fig. 3.6: Interest.java
2  // Calculating compound interest
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class Interest extends Applet {
7     public void paint( Graphics g )
8     {
9        double amount, principal = 1000.0, rate = .05;
10       int yPos = 40;
11
12       g.drawString( "Year", 25, 25 );
13       g.drawString( "Amount on deposit", 100, 25 );
14
15       for ( int year = 1; year <= 10; year++ ) {
16          amount = principal * Math.pow( 1.0 + rate, year );
17          g.drawString( Integer.toString( year ), 25, yPos );
18         g.drawString( Double.toString( amount ), 100, yPos );
19          yPos += 15;
20       }
21    }
22 }
```

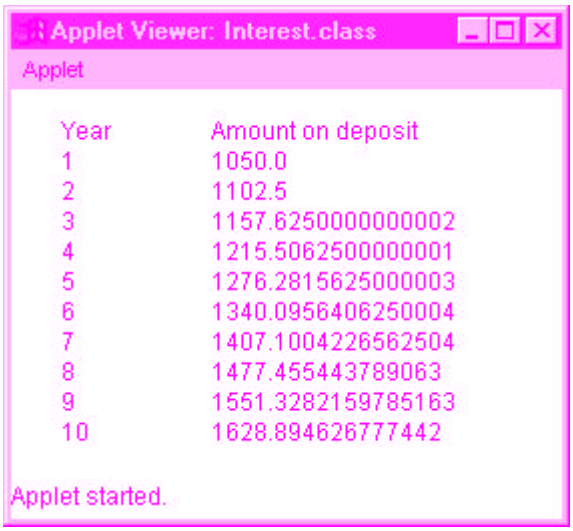


Fig. 3.6 Calculating compound interest with `for`.

```
1  // Fig. 3.7: SwitchTest.java
2  // Counting letter grades
3  import java.awt.*;
4  import java.applet.Applet;
5  import java.awt.event.*;
6
7  public class SwitchTest extends Applet
8                implements ActionListener {
9     Label prompt;      // label for TextField
10    TextField input;   // TextField to enter grades
11
12    int aCount = 0, bCount = 0, cCount = 0,
13        dCount = 0, fCount = 0;
14
15    public void init()
16    {
17       prompt = new Label( "Enter grade" );
18       input = new TextField( 2 );
19       input.addActionListener( this );
20       add( prompt );
```

Fig. 3.7 An example using **`switch`** (part 1 of 3).

```
21       add( input );
22    }
23
24    public void paint( Graphics g )
25    {
26       g.drawString( "Totals for each letter grade:", 25, 40 );
27       g.drawString( "A: " + aCount, 25, 55 );
28       g.drawString( "B: " + bCount, 25, 70 );
29       g.drawString( "C: " + cCount, 25, 85 );
30       g.drawString( "D: " + dCount, 25, 100 );
31       g.drawString( "F: " + fCount, 25, 115 );
32    }
33
34    public void actionPerformed( ActionEvent e )
35    {
36       String val = e.getActionCommand();
37       char grade = val.charAt( 0 );
38
39       showStatus( "" );       // clear status bar area
40       input.setText( "" );    // clear input TextField
41
42       switch ( grade ) {
43
44          case 'A': case 'a':  // Grade was uppercase A
45             ++aCount;         // or lowercase a.
46             break;
47
48          case 'B': case 'b':  // Grade was uppercase B
49             ++bCount;         // or lowercase b.
50             break;
51
52          case 'C': case 'c':  // Grade was uppercase C
53             ++cCount;         // or lowercase c.
54             break;
55
56          case 'D': case 'd':  // Grade was uppercase D
57             ++dCount;         // or lowercase d.
58             break;
59
```

```
60            case 'F': case 'f':  // Grade was uppercase F
61               ++fCount;         // or lowercase f.
62               break;
63
64            default:             // catch all other characters
65               showStatus( "Incorrect grade. Enter new grade." );
66               break;
67         }
68
69         repaint();  // display summary of results
70      }
71   }
```

Fig. 3.7 An example using **`switch`** (part 2 of 3).

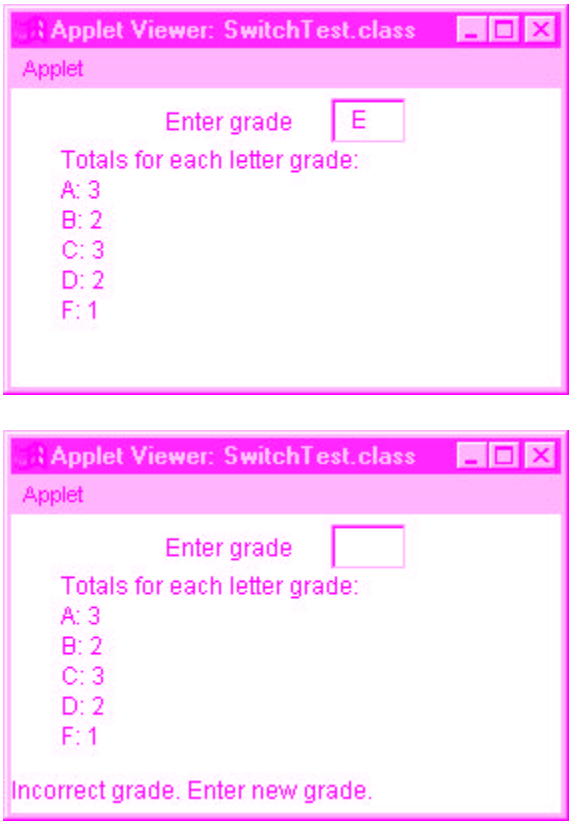


Fig. 3.7 An example using **`switch`** (part 3 of 3).

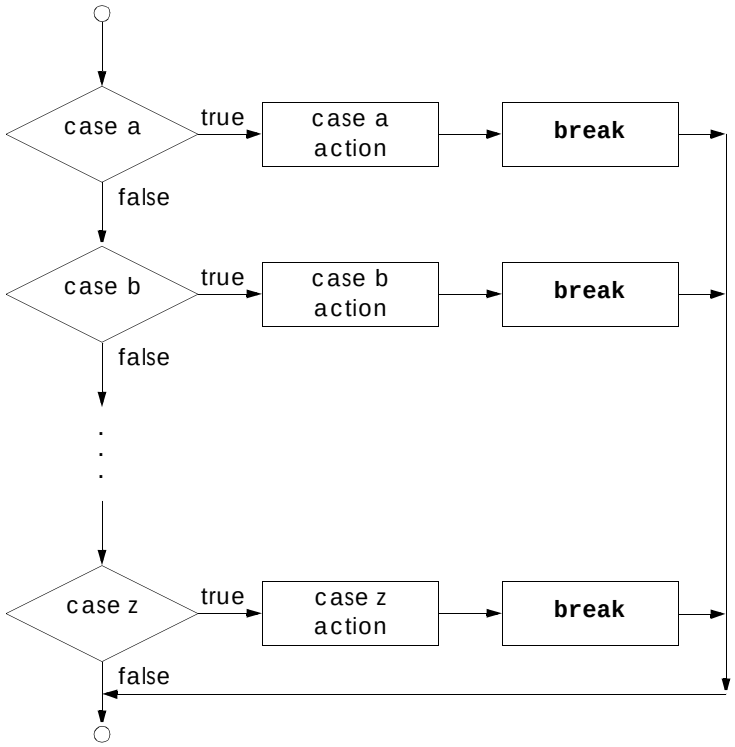


Fig. 3.8 The **switch** multiple-selection structure.

```
1  // Fig. 3.9: DoWhileTest.java
2  // Using the do/while repetition structure
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class DoWhileTest extends Applet {
7     public void paint( Graphics g )
8     {
9        int counter = 1;
10       int xPos = 25;
11
12       do {
13          g.drawString( Integer.toString( counter ), xPos, 25 );
14          xPos += 15;
15       } while ( ++counter <= 10 );
16    }
17 }
```

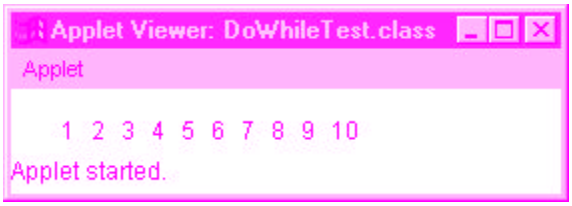


Fig. 3.9 Using the **do/while** repetition structure.

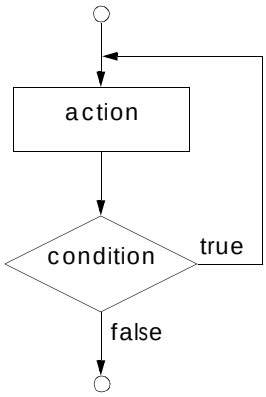


Fig. 3.10 Flowcharting the **do/while** repetition structure.

.

```
1  // Fig. 3.11: BreakTest.java
2  // Using the break statement in a for structure
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class BreakTest extends Applet {
7     public void paint( Graphics g )
8     {
9        int count, xPos = 25;
10
11       for ( count = 1; count <= 10; count++ ) {
12          if ( count == 5 )
13             break;  // break loop only if count == 5
14
15          g.drawString( Integer.toString( count ), xPos, 25 );
16          xPos += 10;
17       }
18
19       g.drawString( "Broke out of loop at count = " + count,
20                     25, 40 );
```

Fig. 3.11 Using the **break** statement in a **for** structure (part 1 of 2).

```
21      }
22   }
```

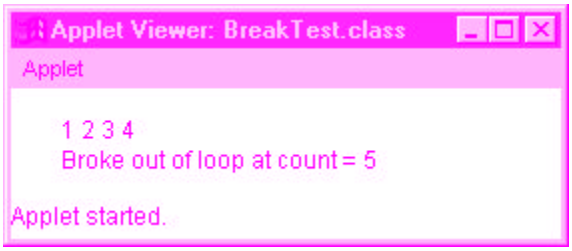


Fig. 3.11 Using the **break** statement in a **for** structure (part 2 of 2).

.

```
1   // Fig. 3.12: ContinueTest.java
2   // Using the continue statement in a for structure
3   import java.awt.Graphics;
4   import java.applet.Applet;
5
6   public class ContinueTest extends Applet {
7      public void paint( Graphics g )
8      {
9         int xPos = 25;
10
11        for ( int count = 1; count <= 10; count++ ) {
12           if ( count == 5 )
13              continue;  // skip remaining code in loop
14                         // only if count == 5
15
16           g.drawString( Integer.toString( count ), xPos, 25 );
17           xPos += 10;
18        }
19
20        g.drawString( "Used continue to skip printing 5",
21                      25, 40 );
22     }
23  }
```

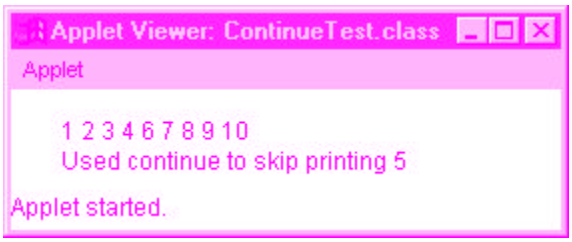


Fig. 3.12 Using the **continue** statement in a **for** structure .

```
1  // Fig. 3.13: BreakLabelTest.java
2  // Using the break statement with a label
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class BreakLabelTest extends Applet {
7     public void paint( Graphics g )
8     {
9        int xPos, yPos = 0;
10
11       stop: {  // labeled compound statement
12          for ( int row = 1; row <= 10; row++ ) {
13             xPos = 25;
14             yPos += 15;
15
16             for ( int column = 1; column <= 5 ; column++ ) {
17
18                if ( row == 5 )
19                   break stop; // jump to end of stop block
20
21                g.drawString( "#", xPos, yPos );
22                xPos += 7;
23             }
24          }
25
26          yPos += 15;
27          g.drawString( "Loops terminated normally", 25, yPos );
28       }
29
30       yPos += 15;
31       g.drawString( "End of paint method", 25, yPos );
32    }
33 }
```

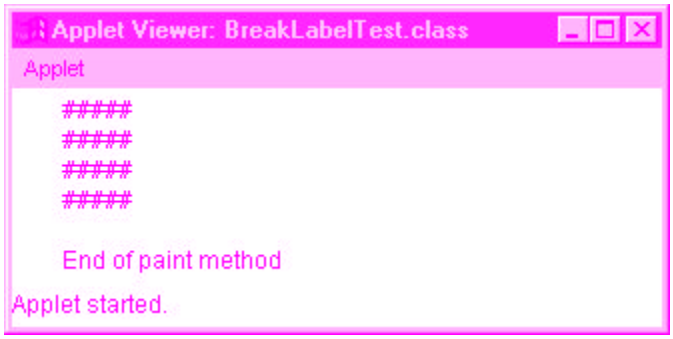


Fig. 3.13 Using a labeled **break** statement in a nested **for** structure.

```
1  // Fig. 3.14: ContinueLabelTest.java
2  // Using the continue statement with a label
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class ContinueLabelTest extends Applet {
7     public void paint( Graphics g )
8     {
9        int xPos, yPos = 0;
10
11       nextRow:   // target label of continue statement
12          for ( int row = 1; row <= 5; row++ ) {
13             xPos = 25;
14             yPos += 15;
15
16             for ( int column = 1; column <= 10; column++ ) {
17
18                if ( column > row )
19                   continue nextRow; // next iteration of
20                                     // labeled loop
21
22                g.drawString( "#", xPos, yPos );
23                xPos += 7;
24             }
25          }
26    }
27 }
```

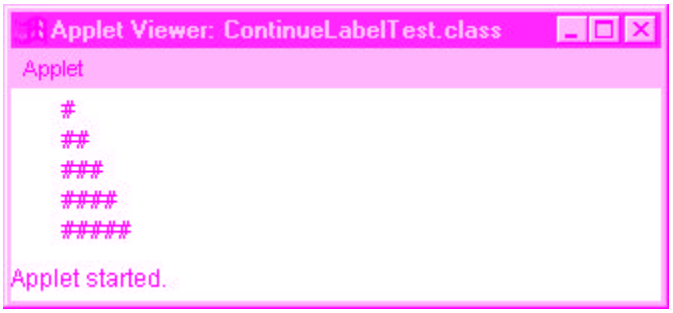


Fig. 3.14 Using a labeled **continue** statement in a nested **for** structure

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

Fig. 3.15 Truth table for the **&&** (logical AND) operator.

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

Fig. 3.16 Truth table for the logical OR (||) operator.

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

Fig. 3.17 Truth table for the boolean logical exclusive OR (^) operator.

expression	!expression
false	true
true	false

Fig. 3.18 Truth table for operator ! (logical NOT).

```
1  // Fig. 3.19: LogicalOperators.java
2  // Demonstrating the logical operators
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class LogicalOperators extends Applet {
7     public void paint( Graphics g )
8     {
9        g.drawString( "Logical AND (&&)", 10, 25 );
```

Fig. 3.19 Demonstrating the logical operators (part 1 of 3).

```
10          g.drawString( "F && F: " + ( false && false ), 10, 40 );
11          g.drawString( "F && T: " + ( false && true ), 10, 55 );
12          g.drawString( "T && F: " + ( true && false ), 10, 70 );
13          g.drawString( "T && T: " + ( true && true ), 10, 85 );
14
15          g.drawString( "Logical OR (||)", 215, 25 );
16          g.drawString( "F || F: " + ( false || false ), 215, 40 );
17          g.drawString( "F || T: " + ( false || true ), 215, 55 );
18          g.drawString( "T || F: " + ( true || false ), 215, 70 );
19          g.drawString( "T || T: " + ( true || true ), 215, 85 );
20
21          g.drawString( "Boolean logical AND (&)", 10, 115 );
22          g.drawString( "F & F: " + ( false & false ), 10, 130 );
23          g.drawString( "F & T: " + ( false & true ), 10, 145 );
24          g.drawString( "T & F: " + ( true & false ), 10, 160 );
25          g.drawString( "T & T: " + ( true & true ), 10, 175 );
26
27          g.drawString( "Boolean logical inclusive OR (|)",
28                        215, 115 );
29          g.drawString( "F | F: " + ( false | false ), 215, 130 );
30          g.drawString( "F | T: " + ( false | true ), 215, 145 );
31          g.drawString( "T | F: " + ( true | false ), 215, 160 );
32          g.drawString( "T | T: " + ( true | true ), 215, 175 );
33
34          g.drawString( "Boolean logical exclusive OR (^)",
35                        10, 205 );
36          g.drawString( "F ^ F: " + ( false ^ false ), 10, 220 );
37          g.drawString( "F ^ T: " + ( false ^ true ), 10, 235 );
38          g.drawString( "T ^ F: " + ( true ^ false ), 10, 250 );
39          g.drawString( "T ^ T: " + ( true ^ true ), 10, 265 );
40
41          g.drawString( "Logical NOT (!)",
42                        215, 205 );
43          g.drawString( "!F: " + ( !false ), 215, 220 );
44          g.drawString( "!T: " + ( !true ), 215, 235 );
```

Fig. 3.19 Demonstrating the logical operators (part 2 of 3).

```
45      }
46   }
```

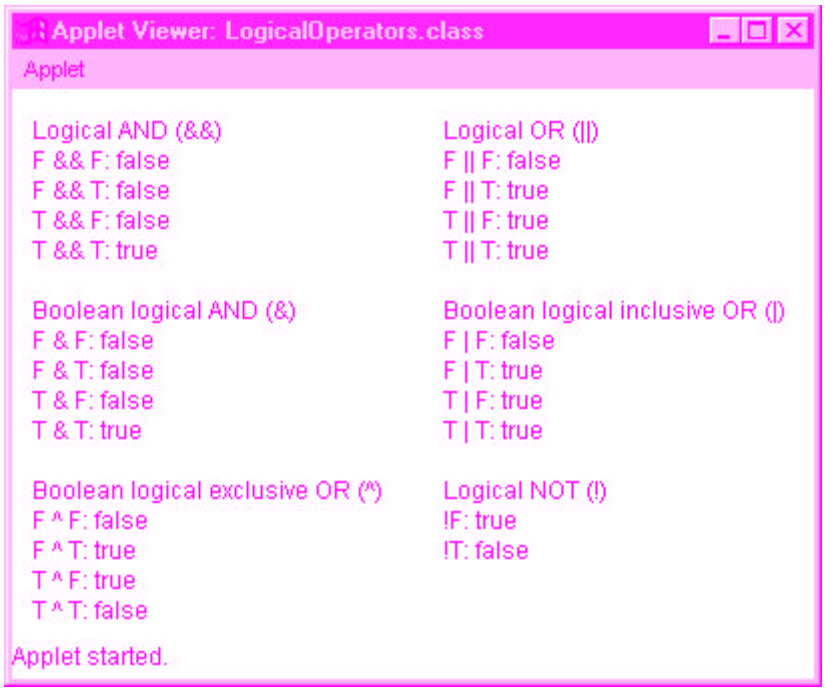


Fig. 3.19 Demonstrating the logical operators (part 3 of 3).

Operators	Associativity	Type
`( )`	left to right	parentheses
`++ -- + - ! (`*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

Fig. 3.20 Operator precedence and associativity.

Operators	Associativity	Type
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 3.20 Operator precedence and associativity.

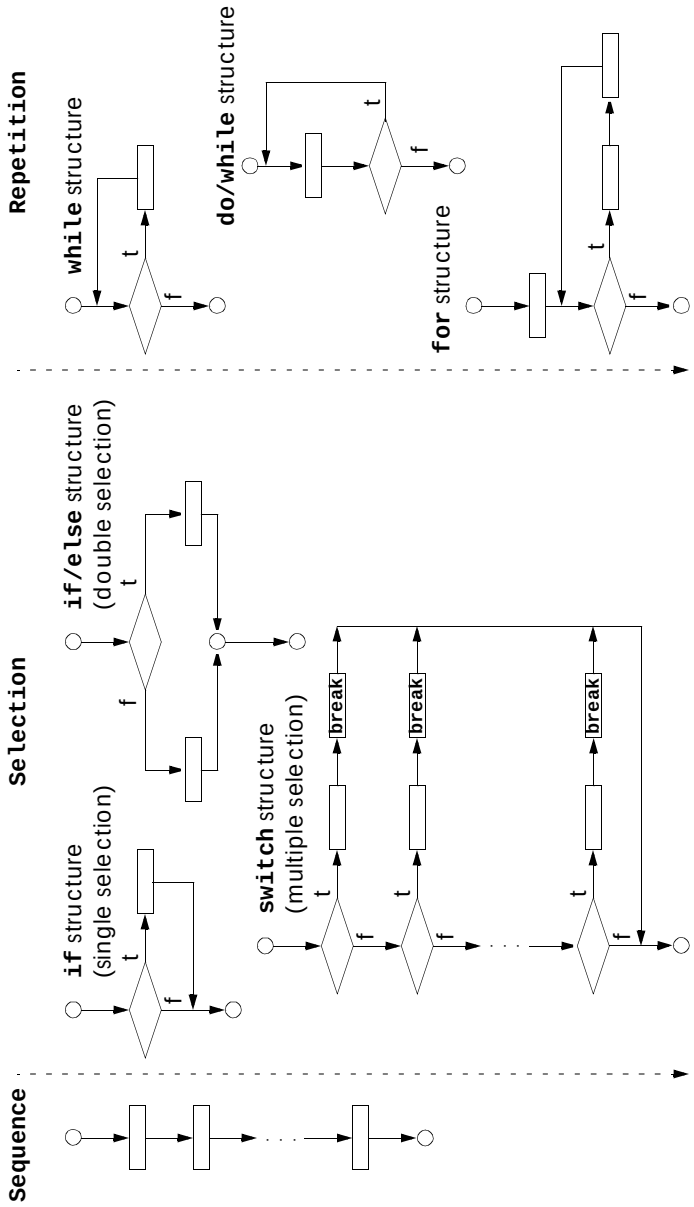


Fig. 3.21 Java's single-entry/single-exit sequence, selection, and repetition structures.

Rules for Forming Structured Programs

1) Begin with the "simplest flowchart" (Fig. 3.23).

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. 3.22 Rules for forming structured programs.

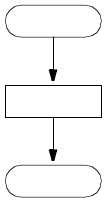


Fig. 3.23 The simplest flowchart.

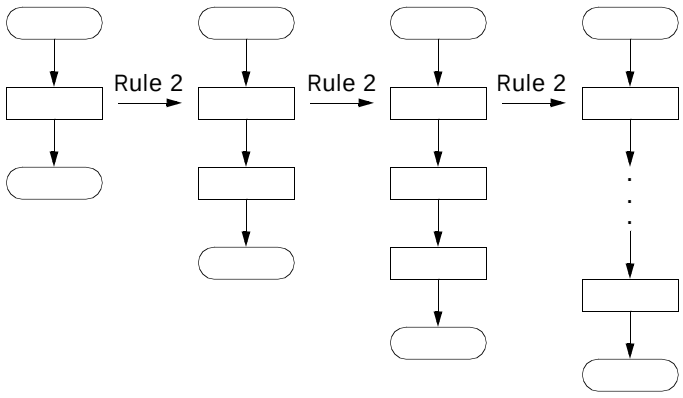


Fig. 3.24 Repeatedly applying rule 2 of Fig. 3.22 to the simplest flowchart.

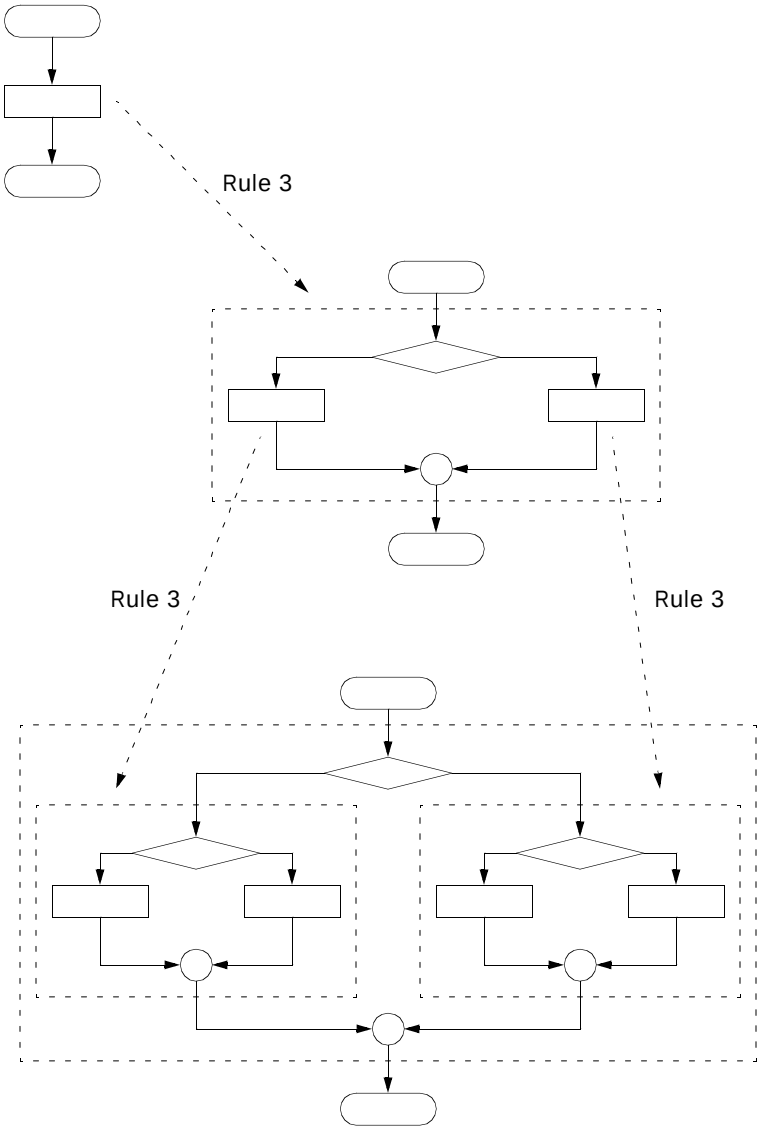


Fig. 3.25 Applying rule 3 of Fig. 3.22 to the simplest flowchart.

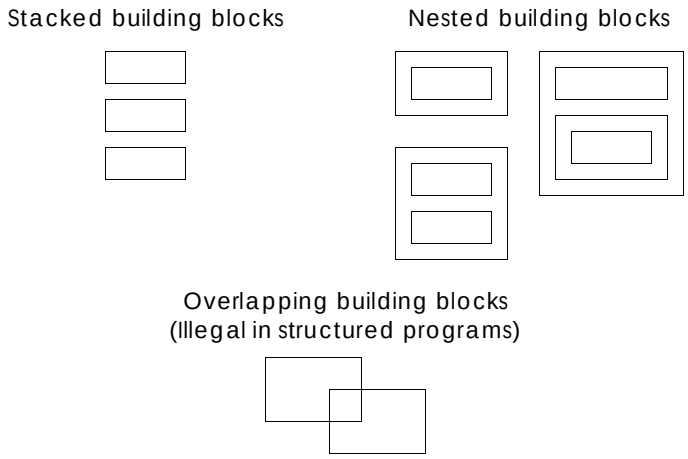


Fig. 3.26 Stacked, nested, and overlapped building blocks.

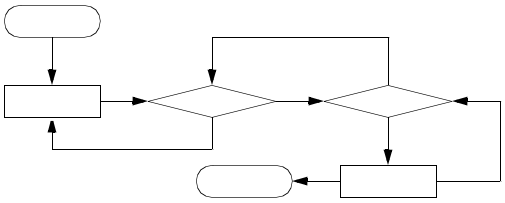


Fig. 3.27 An unstructured flowchart.