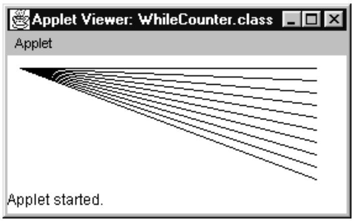
```
// Fig. 5.1: WhileCounter.java
// Counter-controlled repetition
import java.awt.Graphics;
import javax.swing.JApplet;

public class WhileCounter extends JApplet {
   public void paint( Graphics g )
   {
      int counter = 1;              // initialization

      while ( counter <= 10 ) {     // repetition condition
         g.drawLine( 10, 10, 250, counter * 10 );
         ++counter;                 // increment
      }
   }
}
```



Applet started.

Fig. 5.1 Counter-controlled repetition.

```
// Fig. 5.2: ForCounter.java
// Counter-controlled repetition with the for structure
import java.awt.Graphics;
import javax.swing.JApplet;

public class ForCounter extends JApplet {
   public void paint( Graphics g )
   {
      // Initialization, repetition condition and incrementing
      // are all included in the for structure header.
      for ( int counter = 1; counter <= 10; counter++ )
         g.drawLine( 10, 10, 250, counter * 10 );
   }
}
```

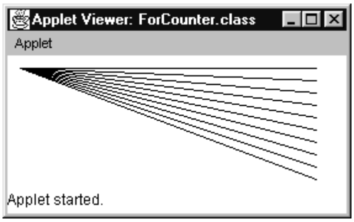


Fig. 5.2 Counter-controlled repetition with the **for** structure.

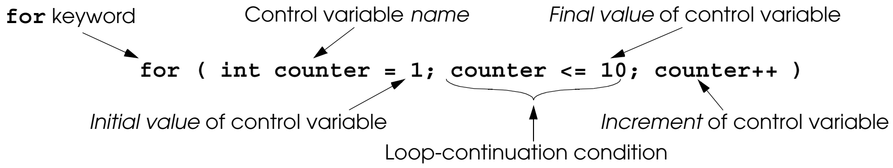

for keyword
Control variable *name*
Final value of control variable
for (int counter = 1; counter <= 10; counter++)
Initial value of control variable
Increment of control variable
Loop-continuation condition

Fig. 5.3 Components of a typical `for` header.

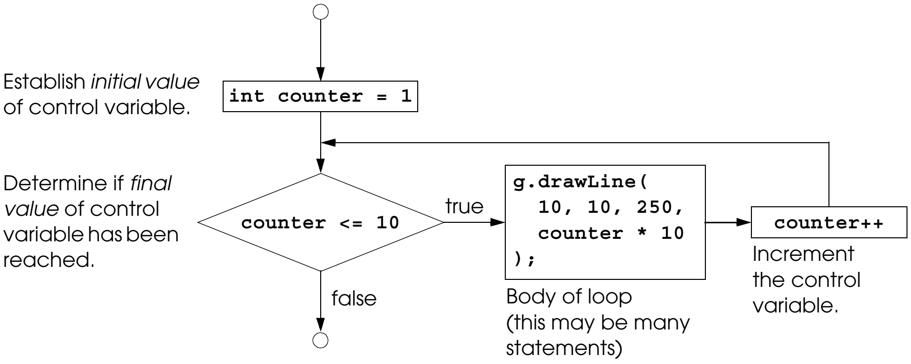
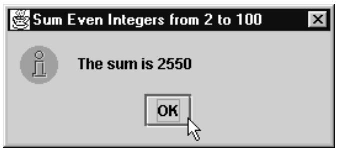

Establish *initial value* of control variable.
int counter = 1
Determine if *final value* of control variable has been reached.
counter <= 10
true
false
g.drawLine(
10, 10, 250,
counter * 10
);
Body of loop (this may be many statements)
counter++
Increment the control variable.

Fig. 5.4 Flowcharting a typical `for` repetition structure.

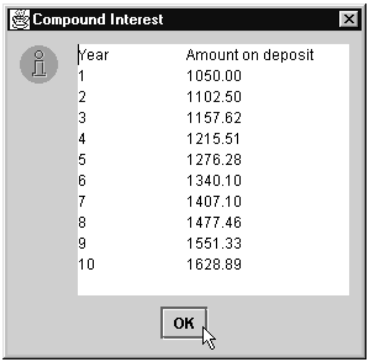
```
// Fig. 5.5: Sum.java
// Counter-controlled repetition with the for structure
import javax.swing.JOptionPane;

public class Sum {
   public static void main( String args[] )
   {
      int sum = 0;

      for ( int number = 2; number <= 100; number += 2 )
         sum += number;

      JOptionPane.showMessageDialog( null,
         "The sum is " + sum,
         "Sum Even Integers from 2 to 100",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );   // terminate the application
   }
}
```



The sum is 2550

Fig. 5.5 Summation with **for**.

```java
// Fig. 5.6: Interest.java
// Calculating compound interest
import java.text.DecimalFormat;
import javax.swing.JOptionPane;
import javax.swing.JTextArea;

public class Interest {
   public static void main( String args[] )
   {
      double amount, principal = 1000.0, rate = .05;

      DecimalFormat precisionTwo = new DecimalFormat( "0.00" );
      JTextArea outputTextArea = new JTextArea( 11, 20 );

      outputTextArea.append( "Year\tAmount on deposit\n" );

      for ( int year = 1; year <= 10; year++ ) {
         amount = principal * Math.pow( 1.0 + rate, year );
         outputTextArea.append( year + "\t" +
            precisionTwo.format( amount ) + "\n" );
      }

      JOptionPane.showMessageDialog(
         null, outputTextArea, "Compound Interest",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );  // terminate the application
   }
}
```

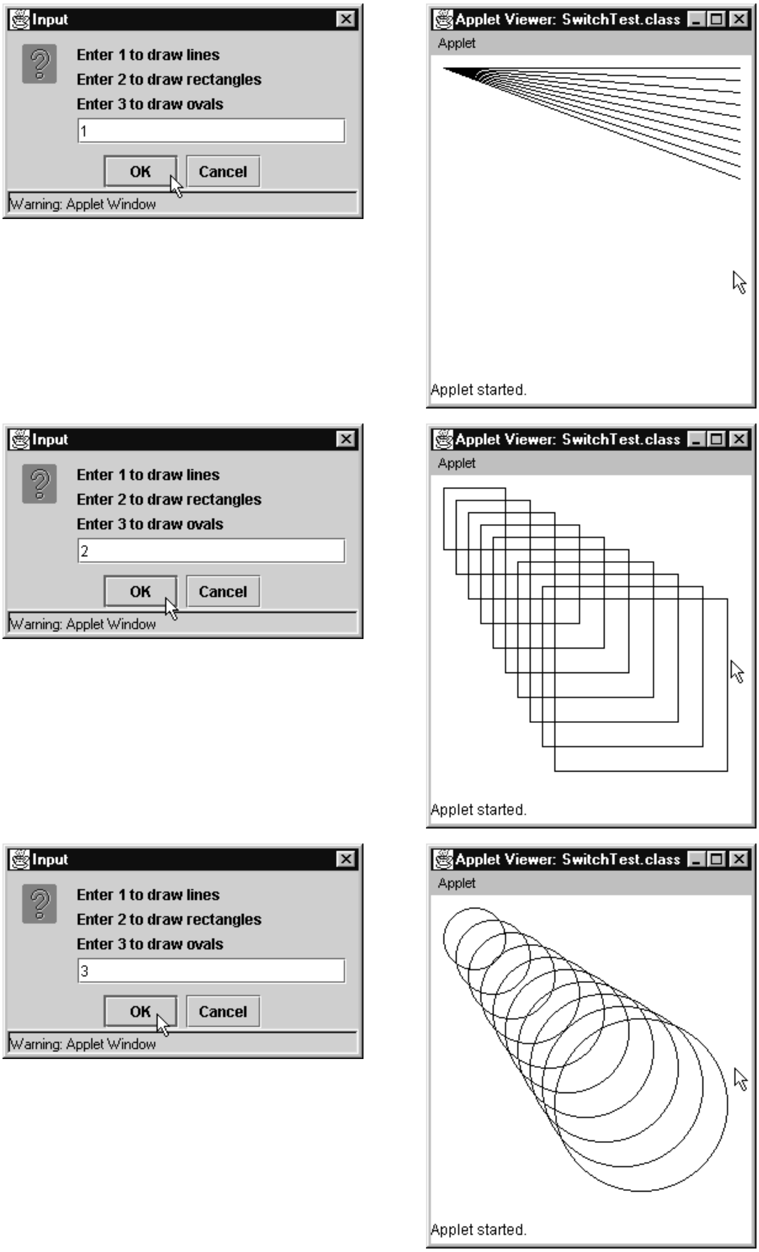


```
Year          Amount on deposit
1             1050.00
2             1102.50
3             1157.62
4             1215.51
5             1276.28
6             1340.10
7             1407.10
8             1477.46
9             1551.33
10            1628.89
```

Fig. 5.6 Calculating compound interest with `for`.

```
// Fig. 5.7: SwitchTest.java
// Drawing shapes
import java.awt.Graphics;
import javax.swing.*;

public class SwitchTest extends JApplet {
   int choice;

   public void init()
   {
      String input;

      input = JOptionPane.showInputDialog(
                 "Enter 1 to draw lines\n" +
                 "Enter 2 to draw rectangles\n" +
                 "Enter 3 to draw ovals\n" );

      choice = Integer.parseInt( input );
   }

   public void paint( Graphics g )
   {
      for ( int i = 0; i < 10; i++ ) {
         switch( choice ) {
            case 1:
               g.drawLine( 10, 10, 250, 10 + i * 10 );
               break;
            case 2:
               g.drawRect( 10 + i * 10, 10 + i * 10,
                           50 + i * 10, 50 + i * 10 );
               break;
            case 3:
               g.drawOval( 10 + i * 10, 10 + i * 10,
                           50 + i * 10, 50 + i * 10 );
               break;
            default:
               JOptionPane.showMessageDialog(
                  null, "Invalid value entered" );
         } // end switch
      } // end for
   } // end paint()
} // end class SwitchTest
```

Fig. 5.7 An example using **switch** (part 1 of 2).



Input

Enter 1 to draw lines
Enter 2 to draw rectangles
Enter 3 to draw ovals

1

Warning: Applet Window

Applet Viewer: SwitchTest.class

Applet started.

Input

Enter 1 to draw lines
Enter 2 to draw rectangles
Enter 3 to draw ovals

2

Warning: Applet Window

Applet Viewer: SwitchTest.class

Applet started.

Input

Enter 1 to draw lines
Enter 2 to draw rectangles
Enter 3 to draw ovals

3

Warning: Applet Window

Applet Viewer: SwitchTest.class

Applet started.

Fig. 5.7 An example using **`switch`** (part 2 of 2).

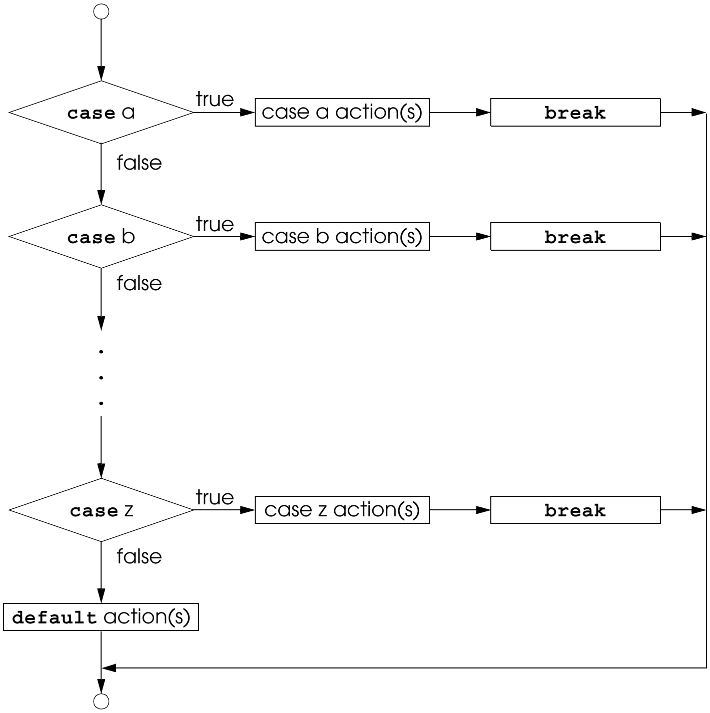


Fig. 5.8 The `switch` multiple-selection structure.

```java
// Fig. 5.9: DoWhileTest.java
// Using the do/while repetition structure
import java.awt.Graphics;
import javax.swing.JApplet;

public class DoWhileTest extends JApplet {
   public void paint( Graphics g )
   {
      int counter = 1;

      do {
         g.drawOval( 110 - counter * 10, 110 - counter * 10,
                     counter * 20, counter * 20 );
         ++counter;
      } while ( counter <= 10 );
   }
}
```

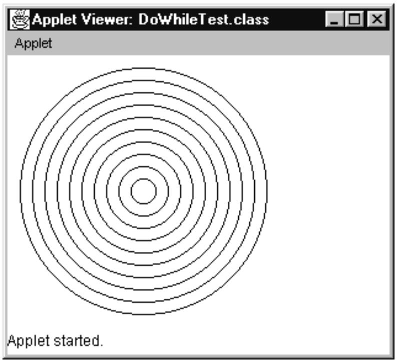


Fig. 5.9 Using the **do/while** repetition structure.

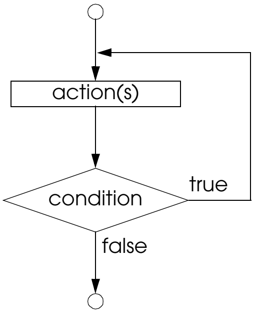
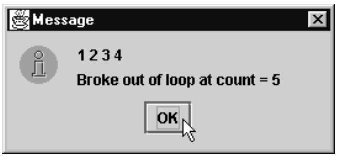


Fig. 5.10 Flowcharting the **`do/while`** repetition structure.

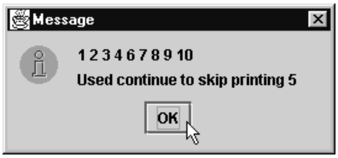
```
// Fig. 5.11: BreakTest.java
// Using the break statement in a for structure
import javax.swing.JOptionPane;

public class BreakTest {
   public static void main( String args[] )
   {
      String output = "";
      int count;

      for ( count = 1; count <= 10; count++ ) {
         if ( count == 5 )
            break;  // break loop only if count == 5

         output += count + " ";
      }

      output += "\nBroke out of loop at count = " + count;
      JOptionPane.showMessageDialog( null, output );
      System.exit( 0 );
   }
}
```



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

Fig. 5.11 Using the **`break`** statement in a **`for`** structure.

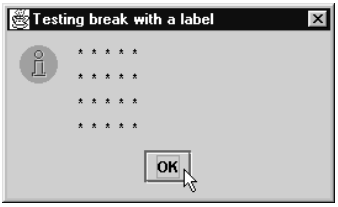
```
// Fig. 5.12: ContinueTest.java
// Using the continue statement in a for structure
import javax.swing.JOptionPane;

public class ContinueTest {
   public static void main( String args[] )
   {
      String output = "";

      for ( int count = 1; count <= 10; count++ ) {
         if ( count == 5 )
            continue;  // skip remaining code in loop
                       // only if count == 5

         output += count + " ";
      }

      output += "\nUsed continue to skip printing 5";
      JOptionPane.showMessageDialog( null, output );
      System.exit( 0 );
   }
}
```



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

Fig. 5.12 Using the **continue** statement in a **for** structure.

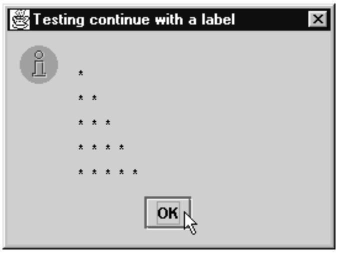
```java
// Fig. 5.13: BreakLabelTest.java
// Using the break statement with a label
import javax.swing.JOptionPane;

public class BreakLabelTest {
   public static void main( String args[] )
   {
      String output = "";

      stop: {   // labeled compound statement
         for ( int row = 1; row <= 10; row++ ) {
            for ( int column = 1; column <= 5 ; column++ ) {

               if ( row == 5 )
                  break stop; // jump to end of stop block

               output += "*  ";
            }

            output += "\n";
         }

         // the following line is skipped
         output += "\nLoops terminated normally";
      }

      JOptionPane.showMessageDialog(
         null, output,"Testing break with a label",
         JOptionPane.INFORMATION_MESSAGE );
      System.exit( 0 );
   }
}
```



```
* * * * *
* * * * *
* * * * *
* * * * *
```

Fig. 5.13 Using a labeled **break** statement in a nested **for** structure.

```
// Fig. 5.14: ContinueLabelTest.java
// Using the continue statement with a label
import javax.swing.JOptionPane;

public class ContinueLabelTest {
   public static void main( String args[] )
   {
      String output = "";

      nextRow:   // target label of continue statement
         for ( int row = 1; row <= 5; row++ ) {
            output += "\n";

            for ( int column = 1; column <= 10; column++ ) {

               if ( column > row )
                  continue nextRow; // next iteration of
                                    // labeled loop

               output += "*  ";
            }
         }

      JOptionPane.showMessageDialog(
         null, output,"Testing continue with a label",
         JOptionPane.INFORMATION_MESSAGE );
      System.exit( 0 );
   }
}
```



```
*
* *
* * *
* * * *
* * * * *
```

Fig. 5.14 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. 5.15 Truth table for the **&&** (logical AND) operator.

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

Fig. 5.16 Truth table for the `||` (logical OR) operator.

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

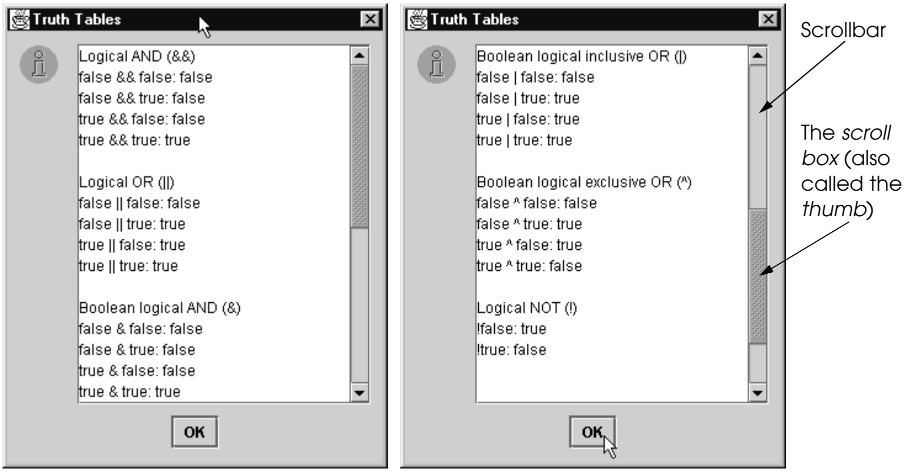
Fig. 5.17 Truth table for the boolean logical exclusive OR (^) operator.

expression	! expression
`false`	`true`
`true`	`false`

Fig. 5.18 Truth table for operator **!** (logical NOT).

```java
// Fig. 5.19: LogicalOperators.java
// Demonstrating the logical operators
import javax.swing.*;

public class LogicalOperators {
   public static void main( String args[] )
   {
      JTextArea outputArea = new JTextArea( 17, 20 );
      JScrollPane scroller = new JScrollPane( outputArea );
      String output = "";

      output += "Logical AND (&&)" +
                "\nfalse && false: " + ( false && false ) +
                "\nfalse && true: " + ( false && true ) +
                "\ntrue && false: " + ( true && false ) +
                "\ntrue && true: " + ( true && true );

      output += "\n\nLogical OR (||)" +
                "\nfalse || false: " + ( false || false ) +
                "\nfalse || true: " + ( false || true ) +
                "\ntrue || false: " + ( true || false ) +
                "\ntrue || true: " + ( true || true );

      output += "\n\nBoolean logical AND (&)" +
                "\nfalse & false: " + ( false & false ) +
                "\nfalse & true: " + ( false & true ) +
                "\ntrue & false: " + ( true & false ) +
                "\ntrue & true: " + ( true & true );

      output += "\n\nBoolean logical inclusive OR (|)" +
                "\nfalse | false: " + ( false | false ) +
                "\nfalse | true: " + ( false | true ) +
                "\ntrue | false: " + ( true | false ) +
                "\ntrue | true: " + ( true | true );

      output += "\n\nBoolean logical exclusive OR (^)" +
                "\nfalse ^ false: " + ( false ^ false ) +
                "\nfalse ^ true: " + ( false ^ true ) +
                "\ntrue ^ false: " + ( true ^ false ) +
                "\ntrue ^ true: " + ( true ^ true );

      output += "\n\nLogical NOT (!)" +
                "\n!false: " + ( !false ) +
                "\n!true: " + ( !true );

      outputArea.setText( output );
      JOptionPane.showMessageDialog( null, scroller,
         "Truth Tables", JOptionPane.INFORMATION_MESSAGE );
      System.exit( 0 );
   }
}
```

Fig. 5.19 Demonstrating the logical operators (part 1 of 2).



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

Logical 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
```

Scrollbar

The *scroll box* (also called the *thumb*)

Fig. 5.19 Demonstrating the logical operators (part 2 of 2).

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.

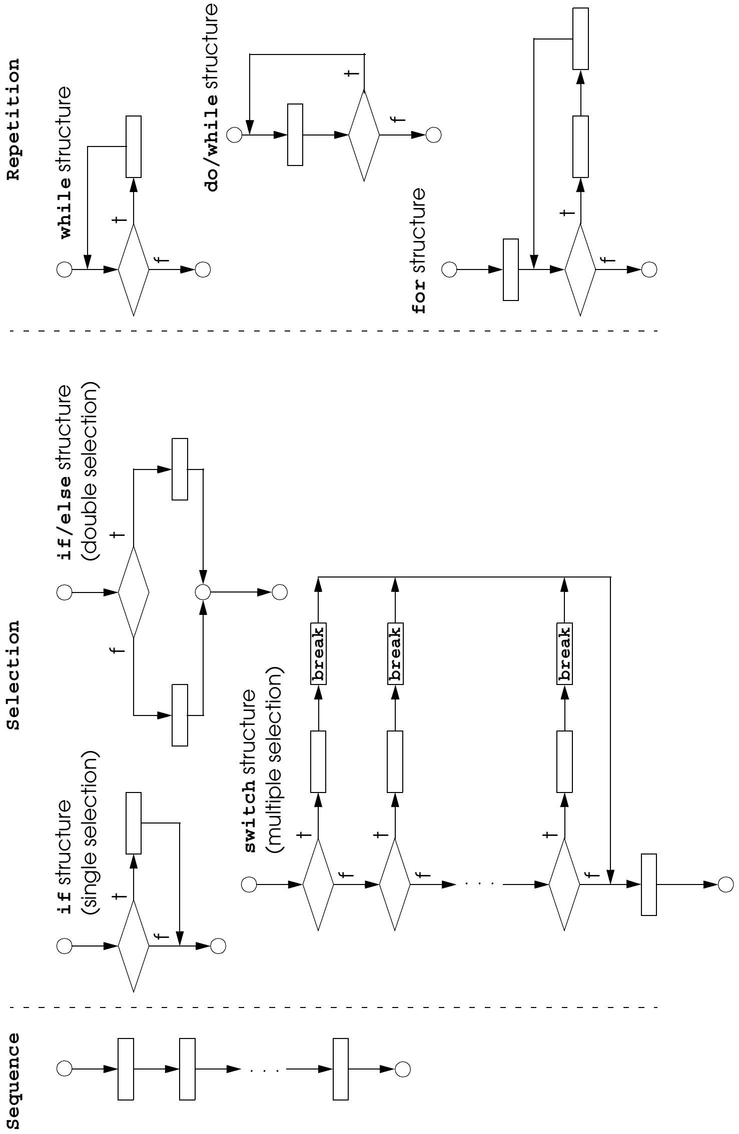


Fig. 5.21 Java's single-entry/single-exit sequence, selection and repetition structures.

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.

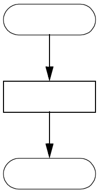


Fig. 5.23 The simplest flowchart.

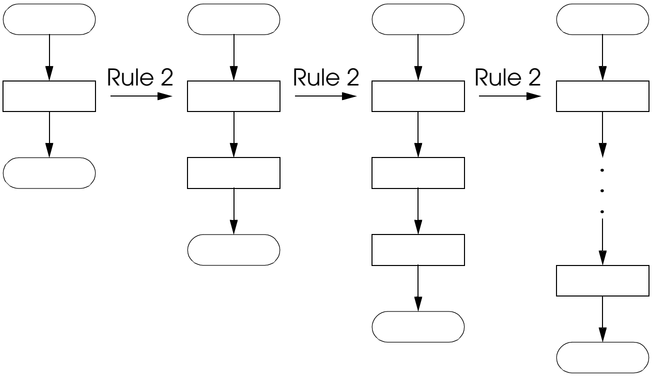


Fig. 5.24 Repeatedly applying rule 2 of Fig. 5.22 to the simplest flowchart.

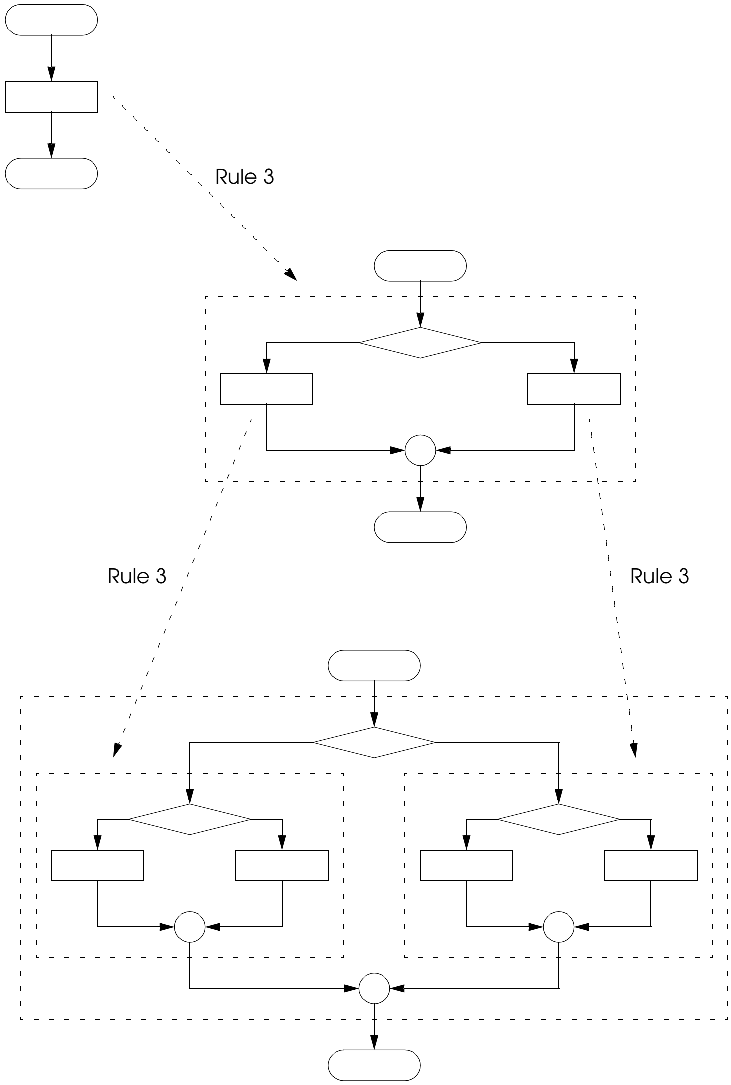


Fig. 5.25 Applying rule 3 of Fig. 5.22 to the simplest flowchart.

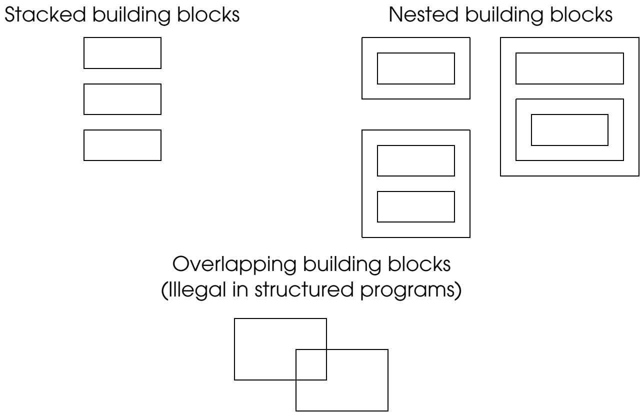


Fig. 5.26 Stacked, nested and overlapped building blocks.

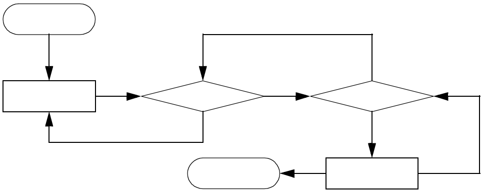


Fig. 5.27 An unstructured flowchart.