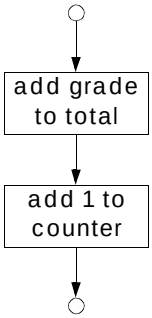
Chapter 2 Developing Java Applications (Main Page)

.




```
total = total + grade;
counter = counter + 1;
```

Fig. 2.1 Flowcharting Java's sequence structure.

Java Keywords

`abstract`	`boolean`	`break`	`byte`	`case`
`catch`	`char`	`class`	`continue`	`default`
`do`	`double`	`else`	`extends`	`false`
`final`	`finally`	`float`	`for`	`if`
`implements`	`import`	`instanceof`	`int`	`interface`
`long`	`native`	`new`	`null`	`package`
`private`	`protected`	`public`	`return`	`short`
`static`	`super`	`switch`	`synchro-nized`	`this`
`throw`	`throws`	`transient`	`true`	`try`
`void`	`volatile`	`while`		

Keywords that are reserved but not used by Java

`const`	`goto`

Fig. 2.2 Java keywords.

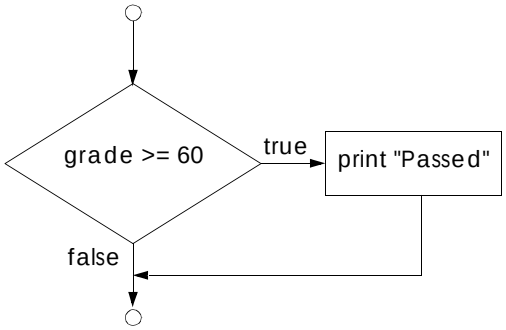


Fig. 2.3 Flowcharting the single-selection `if` structure.

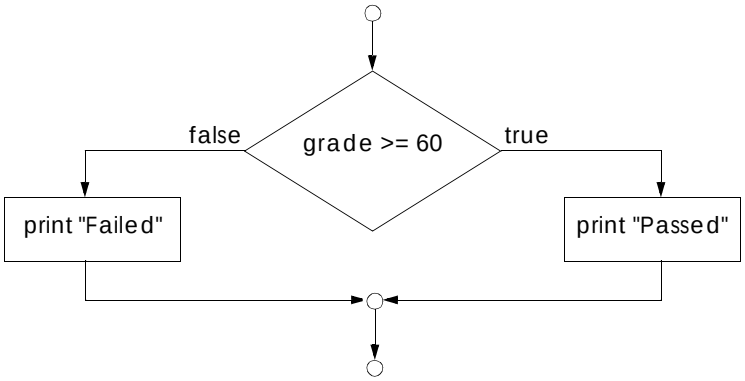


Fig. 2.4 Flowcharting the double-selection `if/else` structure.

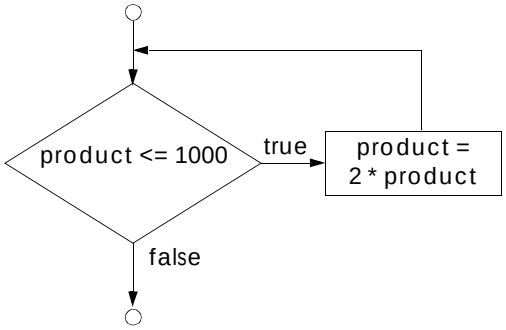


Fig. 2.5 Flowcharting the `while` repetition structure.

Set grade counter to one

While grade counter is less than or equal to ten
Input the next grade

If the letter grade is equal to A
Add grade point value 4 to the total
else if the letter grade is equal to B
Add grade point value 3 to the total
else if the letter grade is equal to C
Add grade point value 2 to the total
else if the letter grade is equal to D
Add grade point value 1 to the total
else if the letter grade is equal to F
Add grade point value 0 to the total

Add one to the grade counter

Set the class average to the total divided by ten
Print the class average

Fig. 2.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

```
1  // Fig. 2.7: Average.java
2  // Class average program with
3  // counter-controlled repetition
4  import java.io.*;
5
6  public class Average {
7     public static void main( String args[] ) throws IOException
8     {
9        int counter, grade, total, average;
10
11       // initialization phase
12       total = 0;
13       counter = 1;
14
15       // processing phase
16       while ( counter <= 10 ) {
17          System.out.print( "Enter letter grade: " );
18          grade = System.in.read();
19
```

Fig. 2.7 Java program for the class-average problem with counter-controlled repetition (part 1 of 2).

```
20          if ( grade == 'A' )
21             total = total + 4;
22          else if ( grade == 'B' )
23             total = total + 3;
24          else if ( grade == 'C' )
25             total = total + 2;
26          else if ( grade == 'D' )
27             total = total + 1;
28
29          System.in.skip( 2 );    // skip the newline character
30          counter = counter + 1;
31       }
32
```

```
33         // termination phase
34         average = total / 10;      // integer division
35         System.out.println( "Class average is " + average );
36      }
37   }
```

```
Enter letter grade: A
Enter letter grade: A
Enter letter grade: A
Enter letter grade: A
Enter letter grade: A
Enter letter grade: B
Enter letter grade: B
Enter letter grade: C
Enter letter grade: D
Enter letter grade: F
Class average is 2
```

Fig. 2.7 Java program for the class-average problem with counter-controlled repetition (part 2 of 2).

Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel value
If the letter grade is equal to A
Add grade point value 4 to the total
else if the letter grade is equal to B
Add grade point value 3 to the total
else if the letter grade is equal to C
Add grade point value 2 to the total
else if the letter grade is equal to D
Add grade point value 1 to the total
else if the letter grade is equal to F
Add grade point value 0 to the total

Add one to the grade counter
Input the next grade (possibly the sentinel)

If the counter is not equal to zero
Set the average to the total divided by the counter
Print the average
else
Print "No grades were entered"

Fig. 2.8 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.

```
1  // Fig. 2.9: Average.java
2  // Class average application with
3  // sentinel-controlled repetition.
4  import java.io.*;
5
6  public class Average {
7     public static void main( String args[] ) throws IOException
8     {
9        double average;  // number with decimal point
10       int counter, grade, total;
11
12       // initialization phase
13       total = 0;
14       counter = 0;
15
16       // processing phase
17       System.out.print( "Enter letter grade, Z to end: " );
18       grade = System.in.read();
19
20       while ( grade != 'Z' ) {
21          if ( grade == 'A' )
22             total = total + 4;
23          else if ( grade == 'B' )
24             total = total + 3;
25          else if ( grade == 'C' )
26             total = total + 2;
27          else if ( grade == 'D' )
28             total = total + 1;
```

```
29
30          System.in.skip( 2 );
31          counter = counter + 1;
32          System.out.print( "Enter letter grade, Z to end: " );
33          grade = System.in.read();
34       }
```

Fig. 2.9 Class-average problem with sentinel-controlled repetition (part 1 of 2).

```
35
36       // termination phase
37       if ( counter != 0 ) {
38          average = (double) total / counter;
39          System.out.println( "Class average is " + average );
40       }
41       else
42          System.out.println( "No grades were entered" );
43    }
44 }
```

```
Enter letter grade, Z to end: A
Enter letter grade, Z to end: A
Enter letter grade, Z to end: A
Enter letter grade, Z to end: A
Enter letter grade, Z to end: A
Enter letter grade, Z to end: B
Enter letter grade, Z to end: B
Enter letter grade, Z to end: B
Enter letter grade, Z to end: B
Enter letter grade, Z to end: B
Enter letter grade, Z to end: Z
Class average is 3.5
```

Fig. 2.9 Class-average problem with sentinel-controlled repetition (part 2 of 2).

Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed

Print "Raise tuition"

Fig. 2.10 Pseudocode for examination-results problem.

```
1  // Fig. 2.11: Analysis.java
2  // Analysis of examination results
3  import java.io.*;
4
5  public class Analysis {
6     public static void main( String args[] ) throws IOException
7     {
8        // initializing variables in declarations
9        int passes = 0, failures = 0, student = 1, result;
10
11       // process 10 students; counter-controlled loop
12       while ( student <= 10 ) {
13          System.out.print( "Enter result (1=pass,2=fail): " );
14          result = System.in.read();
15
16          if ( result == '1' )      // if/else nested in while
17             passes = passes + 1;
18          else
19             failures = failures + 1;
20
21          student = student + 1;
22          System.in.skip( 2 );
23       }
24
25       System.out.println( "Passed " + passes );
26       System.out.println( "Failed " + failures );
27
28       if ( passes > 8 )
29          System.out.println( "Raise tuition " );
30    }
31 }
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Fig. 2.11 Java program and sample execution for examination-results problem (part 1 of 2).

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Fig. 2.11 Java program and sample execution for examination-results problem (part 2 of 2).

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 2.12 Arithmetic assignment operators.

Operator	Called	Sample expression	Explanation
`++`	preincrement	`++a`	Increment **a** by 1 then use the new value of **a** in the expression in which **a** resides.
`++`	postincrement	`a++`	Use the current value of **a** in the expression in which **a** resides, then increment **a** by 1.
`--`	predecrement	`--b`	Decrement **b** by 1 then use the new value of **b** in the expression in which **b** resides.
`--`	postdecrement	`b--`	Use the current value of **b** in the expression in which **b** resides, then decrement **b** by 1.

Fig. 2.13 The increment and decrement operators.

```
1  // Fig. 2.14: Increment.java
2  // Preincrementing and postincrementing
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class Increment extends Applet {
7     public void paint( Graphics g )
8     {
9        int c;
10
11       c = 5;
12       g.drawString( Integer.toString( c ), 25, 25 );
13       g.drawString( Integer.toString( c++ ),  // postincrement
14                     25, 40 );
15       g.drawString( Integer.toString( c ), 25, 55 );
16
```

Fig. 2.14 The difference between preincrementing and postincrementing (part 1 of 2).

```
17       c = 5;
18       g.drawString( Integer.toString( c ), 25, 85 );
19       g.drawString( Integer.toString( ++c ),  // preincrement
20                     25, 100 );
21       g.drawString( Integer.toString( c ), 25, 115 );
22    }
23 }
```

Fig. 2.14 The difference between preincrementing and postincrementing (part 1 of 2).

Operators					Associativity	Type
`()`					left to right	parentheses
`++`	`--`	`+`	`-`	`(`*type*`)`	right to left	unary
`*`	`/`	`%`			left to right	multiplicative

Fig. 2.15 Precedence of the operators encountered so far in the text.

Operators	Associativity	Type
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 2.15 Precedence of the operators encountered so far in the text.

Escape Sequence	Description
\n	Newline. Position the cursor to the beginning of the next line.
\t	Horizontal tab. Move the cursor to the next tab stop.
\r	Carriage return. Position the cursor to the beginning of the current line; do not advance to the next line.
\\	Backslash. Used to print a backslash character.
\'	Single quote. Used to print a single-quote character.
\"	Double quote. Used to print a double-quote character.
\u####	Unicode character. Used to place any Unicode-character constant in a Java program. The #### is a hexadecimal representation of the Unicode value (see Appendix E for information on hexadecimal numbers).

Fig. 2.16 Some common escape sequences.

```
1  // Fig. 2.17: EscapeSequences.java
2  // Demonstrating common escape sequences
3
4  public class EscapeSequences {
5     public static void main( String args[] )
6     {
7        System.out.println( "Displaying single quotes: " +
8                            "\'A\'" );
```

Fig. 2.17 Demonstrating common escape sequences (part 1 of 2).

```
9        System.out.println( "Displaying double quotes: " +
10                           "\"string\"" );
11       System.out.println( "Displaying a backslash: \\" );
12       System.out.println( "Text separated\t\tby two tabs" );
13       System.out.println( "Here is double\n\nspaced text" );
14       System.out.println( "**********\r#####" );
15    }
16 }
```

```
Displaying single quotes: 'A'
Displaying double quotes: "string"
Displaying a backslash: \
Text separated          by two tabs
Here is double

spaced text
#####*****
```