CHAPTER 6

DECISIONS AND REPETITIONS: BASIC ACTIVITIES IN PROGRAMS

So far, our programs have been made of statements that declare variables, assignment statements, input statements, and output statements. Assignment statements, which we studied in the previous chapter, are fundamentally important action statements in computer programs. With an assignment statement, we can copy the contents of one variable into another variable, or we can produce a new value for a variable by inserting an arithmetic expression to the right side of an assignment operation. Assignment statements always write to a location in the main memory of a computer.

In this chapter, we will study more fundamental statements used in computer programming. You will learn to write statements that make decisions (selections) and perform repetitions. Decisions made in a program result in some statements being executed and others not being executed. Performing repetitions means that one or more statements can be executed many times. Java provides **`if`** and **`switch`** statements for making decisions, and **`while`**, **`for`**, and **`do-while`** statements for performing repetitions. In this book, **`if`** statements are usually called **`if`** constructs, and **`while`**, **`for`**, and **`do-while`** statements are called loops.

6.1 Making decisions with keywords if and else

The word "if" in our natural language expresses a condition. We can say: "If the weather is warm and sunny tomorrow, let's go to the beach." The word "if" is used in a similar way in Java. **`if`** is a keyword that identifies the basic decision-making mechanism of Java.

`if` statements, which we often call **`if`** constructs, are used to make decisions in Java. The structure of the simplest **`if`** construct is described in Figure 6-1. An **`if`** construct always contains a *boolean expression* that can be either true or false. Boolean expressions are named after George Boole (1815 - 1864) whose ideas have deeply influenced computing and programming.

Boolean expressions define conditions. In **`if`** constructs, the boolean expression is given in parentheses ( ) after the keyword **`if`**. Every boolean expression has a truth value which is always either true or false, but not both. Provided that the boolean expression is true, the statements inside the braces { } after the boolean expression will be executed. If the boolean expression is false (i.e. not true), the statements inside the braces will not be executed, and the execution of the program continues from the statement that follows the closing brace of the **`if`** construct.

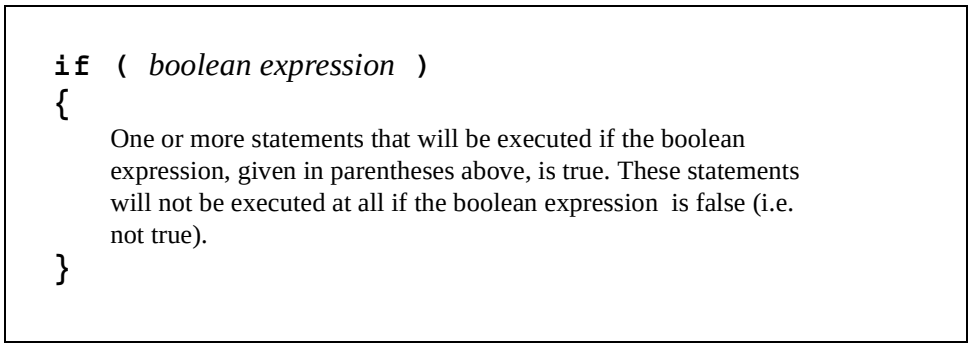


Figure 6-1. The structure of a simple if construct

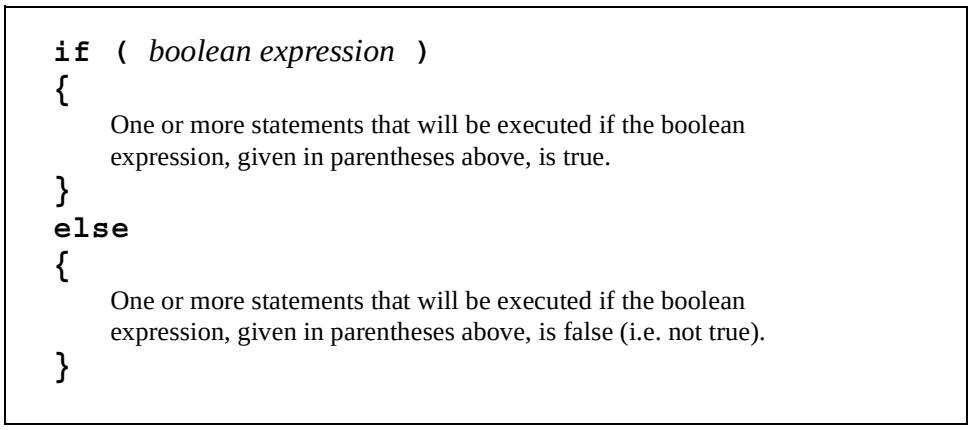


Figure 6-2. The structure of an if-else construct

A more advanced form of **`if`** statement is an **`if-else`** construct which contains two Java keywords, **`if`** and **`else`**. The structure of **`if-else`** construct is explained in Figure 6-2. The **`if-else`** construct has two blocks of statements, and only one block of statements will be executed. When one or more statements are inside braces { }, we can call the group of statements an embraced block of statements, or simply a block. In **`if-else`** constructs, depending on the truth value of the boolean expression, either the first block of statements or the second block, but never both, will be executed. The **`if-else`** construct thus makes a decision as to which program block will be executed.

Program **Largeint.java** is an example where two decisions are made with keywords **`if`** and **`else`**. The first decision is made with an **`if-else`** construct. The second decision is made with a simple **`if`** construct. The program is able to find the largest of three integers that the user types in from the keyboard. First the program decides which is larger of the first two integers. Then it decides whether the third integer is larger than the largest of the first two integers.

Boolean expressions have a truth value, either true or false. To write a boolean expression we need operators that can describe situations that are either true or false. Relational operators, which are listed in Table 6-1, are common in boolean expressions. In program **Largeint.java**, the relational operator < is used in the boolean expression

```
( first_integer  <  second_integer )
```

to test whether it is true that the value of the variable **`first_integer`** is less than the value of the variable **`second_integer`**. A common use for relational operators is to compare values of variables, but relational operators can also take numerical values as operands. For example, the boolean expression

```
( some_variable  ==  0 )
```

tests whether the contents of **`some_variable`** is zero. If the contents of **`some_variable`** is zero, then the expression above is true, otherwise it is false. Because relational operators compare variables and other values, they are also called comparison operators.

Operator ==, like most of the relational operator symbols, consists of two characters. There should be no spaces between the two characters. Writing, for example, = = will result in a compilation error. The compiler would interpret the two equal signs separated with a space as two adjacent assignment operators.

Table 6-1: The relational operators of Java

Operator symbol	Operator name
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal

This program shows that variable declarations do not need to be the first statements of a program. This statement, which is written after a couple of action statements, both declares a variable of type **int** and assigns a value to the declared variable by reading a value from the keyboard. By using these kinds of statements it is possible to make programs somewhat shorter. This single statement means the same as the two statements

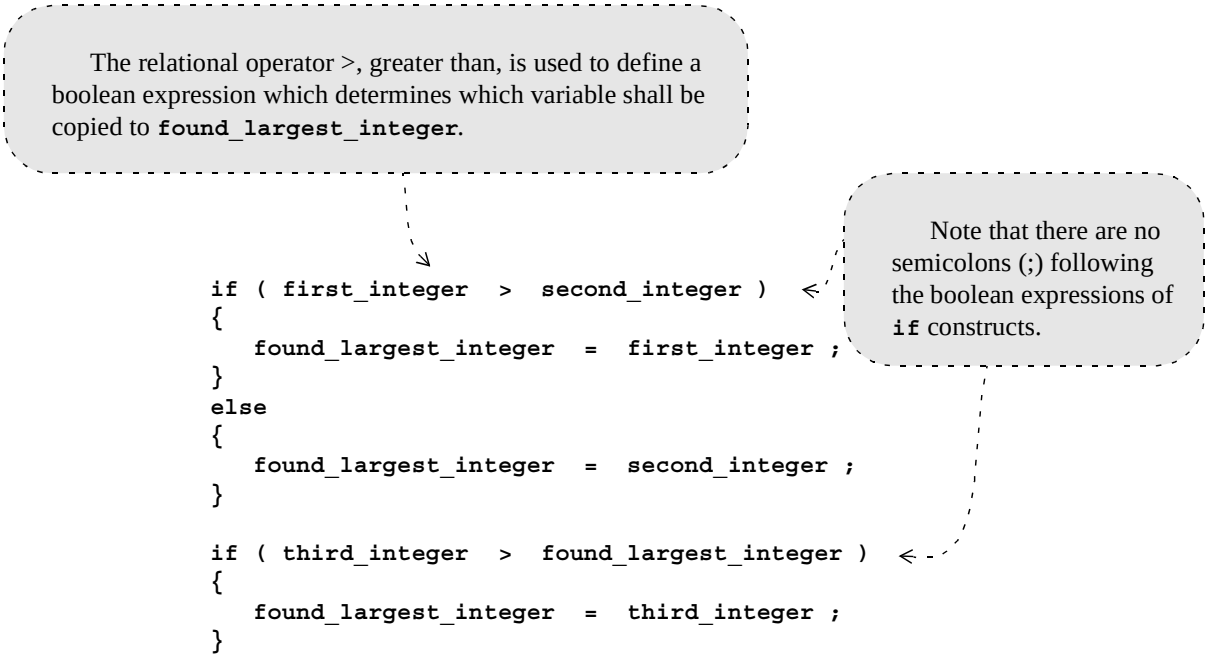
```
int first_integer ;
first_integer = keyboard.nextInt();
```

```
//  Largeint.java

import  java.util.* ;

class Largeint
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      System.out.print( "\n This program can find the largest of three"
                     +  "\n integers you enter from the keyboard. "
                     +  "\n Please, enter three integers separated "
                     +  "\n with spaces :  " ) ;

      int first_integer   =  keyboard.nextInt() ;
      int second_integer  =  keyboard.nextInt() ;
      int third_integer   =  keyboard.nextInt() ;

      int found_largest_integer ;

      if ( first_integer  >  second_integer )
      {
         found_largest_integer  =  first_integer ;
      }
      else
      {
         found_largest_integer  =  second_integer ;
      }

      if ( third_integer  >  found_largest_integer )
      {
         found_largest_integer  =  third_integer ;
      }

      System.out.print( "\n The largest integer is "
                     +  found_largest_integer  +  ".\n" ) ;
   }
}
```

This is the simplest form of an **if** construct, containing only the keyword **if**. If the contents of **third_integer** is greater than the contents of **found_largest_integer**, the contents of variable **third_integer** will be copied to variable **found_largest_integer**.

This is an **if-else** construct. If the contents of **first_integer** is greater than the contents of **second_integer**, the contents of **first_integer** will be copied to **found_largest_integer**. Otherwise, the contents of **second_integer** will be copied to **found_largest_integer**.

Largeint.java - 1.+ A program to find the largest of three integers.



Largeint.java - 1-1. The if constructs that find the largest integer.

```
D:\javafiles2>java Largeint

 This program can find the largest of three
 integers you enter from the keyboard.
 Please, enter three integers separated
 with spaces :  111 222 211

 The largest integer is 222.
```

Largeint.java - X. The program finds 222 to be the largest of 111, 222, and 211.

Program **Evenodd.java** is another example of the use of an **`if-else`** construct. This program asks the user to type in an integer, and it decides whether the given integer is an even or odd number. Because even numbers are equally divisible by two, the **`if-else`** construct in the program **Evenodd.java** simply tests whether it is true that if the given number divided by two would result in a zero remainder. The boolean expression in **Evenodd.java** is

```
( ( integer_from_keyboard  %  2 )  ==  0 )
```

and it contains the arithmetic expression (**`integer_from_keyboard`** % 2). The % operator is an arithmetic operator belonging to the same category as +, -, *, and /. Operator % returns the remainder that would result if its left operand were divided by its right operand. In the expression above, % returns the remainder that would result if the value of variable **`integer_from_keyboard`** were divided by two. The value of the variable is not modified. In the above expression the remainder is calculated first and then the remainder is compared to zero. The relational comparison operator returns true when the remainder is zero. Otherwise it returns false.

When integers are divided by two, the remainder is always zero or one, depending on whether the integer is even or odd. As there are only two possible values for the remainder in divisions by two, the boolean expression

```
( ( integer_from_keyboard  %  2 )  !=  1 )
```

could replace the boolean expression used in the program **Evenodd.java** and the program would work equally well. Operator != returns true when its operands are not equal. Boolean expressions can often be written either with operator == (equal) or with operator != (not equal). When possible, it is better to use operator ==, because it is usually easier to understand.

The remainder operator %

The remainder operator %, which is sometimes called the modulus operator, is used with integers only. Division operations with integers are sometimes inaccurate because computers do not round numbers upwards. For example, the division operation 11/4, eleven divided by four, would be evaluated to 2 although 3 would be closer to the correct value. Computers do not obey human division rules. In division operations involving integers, computers always round downwards. For this reason, the remainder operator % is sometimes useful. To understand operators / and % properly, below are some correct calculations for you to study.

```
1 / 2  is  0          1 % 2  is  1
2 / 2  is  1          2 % 2  is  0
7 / 4  is  1          7 % 4  is  3
9 / 4  is  2          9 % 4  is  1
14 / 5  is  2         14 % 5  is  4
101 / 10 is 10        101 % 10 is 1
```

This program is based on the fact that even numbers are equally divisible by two. (**integer_from_keyboard** % 2) is an arithmetic expression that is part of the boolean expression of this **if** construct. % is the remainder operator of Java. In this case, % returns the remainder that would result if **integer_from_keyboard** were divided by two. With even numbers the remainder is zero, making the boolean expression true.

```
//  Evenodd.java  (c) Kari Laitinen

import java.util.* ;

class Evenodd
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      int  integer_from_keyboard ;

      System.out.print( "\n This program can find out whether an integer"
                    +  "\n is even or odd. Please, enter an integer: " ) ;

      integer_from_keyboard  =  keyboard.nextInt() ;

      if ( ( integer_from_keyboard  %  2 )  ==  0 )
      {
         System.out.print( "\n " + integer_from_keyboard + " is even.\n") ;
      }
      else
      {
         System.out.print( "\n " + integer_from_keyboard + " is odd. \n") ;
      }
   }
}
```

This statement will be executed when the boolean expression, given in parentheses after the keyword **if**, is false.

Evenodd.java - 1. A program to find out whether a given integer is even or odd.

```
D:\javafiles2>java Evenodd

 This program can find out whether an integer
 is even or odd. Please, enter an integer: 12345

 12345 is odd.
```

Evenodd.java - X. The program executed with input value 12345.

In most cases, there is no single way to write a computer program to accomplish the desired program behavior and functionality. This is generally true also for boolean expressions. For example, all the following boolean expressions mean the same

```
( first_integer  <  second_integer )
( second_integer  >  first_integer )
( ( second_integer - first_integer )  >  0 )
( ( first_integer  -  second_integer )  <  0 )
( ( first_integer + 1 )  <=  second_integer )
```

It is, however, good programming practice to try to write simple, non-complex, and elegant programs. For this reason, the last three boolean expressions would not be appropriate choices because they are made unnecessarily complex by using arithmetic operators.

Figure 6-3 shows the structure of a large **`if`** construct, the **`if-else if-else`** construct. Such a construct consists of three parts: an **`if`** part, an **`else if`** part, and an **`else`** part. Each part contains an embraced program block. The execution of the program blocks is controlled by two boolean expressions. To be accurate, the program construct described in Figure 6-3 actually consists of two **`if-else`** constructs. The **`else-if`** part starts the second **`if-else`** construct. The second **`else-if`** construct does not have any braces around itself. You will understand this more clearly when the use of braces in Java program constructs is discussed in more detail in Section 6.6.

The **`if`** structure explained in Figure 6-3 is used in program **Likejava.java**. The **`if-else if-else`** construct selects one of three program blocks to be executed. It is possible to write even more complex **`if`** constructs by adding more **`else if`** parts between the **`if`** part and **`else`** part. Program **Iffing.java** is an example where three **`else if`** parts are used in a single **`if`** construct. In summary, we can state the following facts about **`if`** constructs:

- Every **`if`** construct contains an **`if`** part.
- **`if`** constructs can contain zero, one, or more **`else if`** parts.
- **`if`** constructs contain zero or one **`else`** parts.

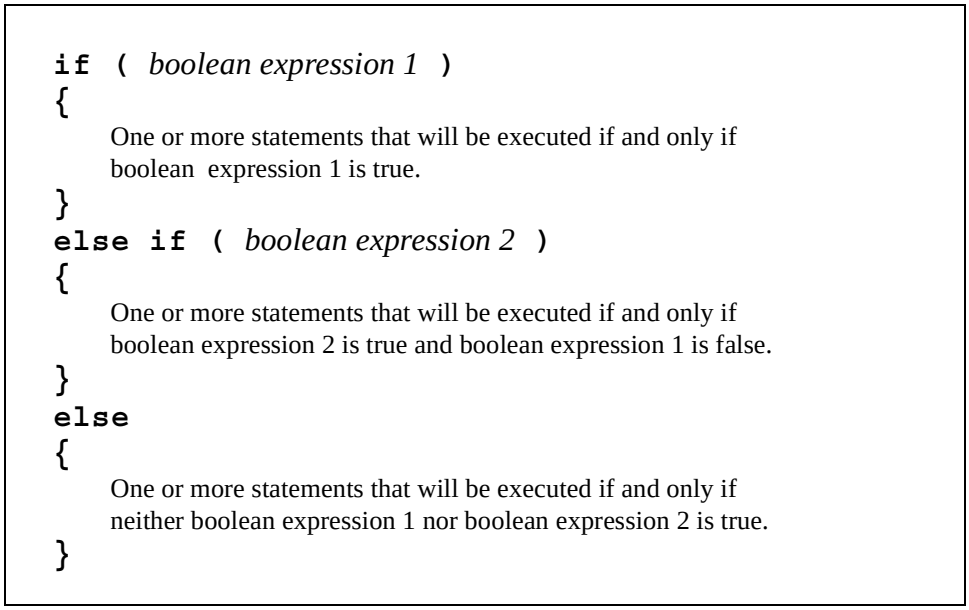


Figure 6-3. The structure of an if-else if-else construct.

Both programs **Likejava.java** and **Iffing.java** investigate what is the character code of the character that the user typed in from the keyboard. Character coding systems were discussed in Chapter 3. Java uses a character coding system called Unicode in which each character is coded with a 16-bit value. In practice, however, the character codes of the English characters as well as many other European characters can be expressed with 8 bits, and the 8 most significant bits of these character codes are zeroes. In Java, single quotes are used to refer to the character codes of characters. For example,

`'Y'` means the character code of Y (89, 59H)

`'n'` means the character code of n (110, 6EH)

`'9'` means the character code of 9 (57, 39H)

`' '` means the character code of space (32, 20H)

By using single quotes it is possible to refer to a character code without remembering the numerical value of the character code. When 'Y' appears in a program, it means the same as the numerical literal constant 89, the character code of uppercase letter Y. For a Java compiler, the relational expression

```
( character_from_keyboard  ==  'Y' )
```

and the relational expression

```
( character_from_keyboard  ==  89 )
```

are technically the same, but for a human reader they are different and the first form is easier to understand. It is difficult to remember the character codes of all characters. Therefore, the meaning of 'Y' is easier to grasp than the meaning of 89. In the terminology of programming languages, characters inside single quotes, such as 'Y', 'n', '9', and ' ', are called character literals. In addition to character literals, we also need string literals like "Hello!" in our programs. In Java, character literals are written with single quotes and string literals are written with double quotes.

Programs **Likejava.java** and **Iffing.java** introduce new operators called logical operators. These operators can combine several relational expressions into a single boolean expression. For example, in the boolean expression

```
( ( character_from_keyboard  ==  'Y' )  ||
  ( character_from_keyboard  ==  'y' )  )
```

the logical-OR operator || combines two relational expressions. The logical-OR operator || returns true if either or both of its operands, the relational expressions, are true. The logical-OR returns false only when both of its operands are false. The truth value of the boolean expression above is evaluated so that the truth values of the relational expressions are evaluated first, and these truth values are then joined with the logical-OR operator ||. If the contents of variable **`character_from_keyboard`** were 121, which is 'y', the relational expressions would have the following truth values

```
( character_from_keyboard  ==  'Y' )    would be false
( character_from_keyboard  ==  'y' )    would be true
```

The truth value of the entire boolean expression would be true because the logical-OR operation result for false and true is true. (To be accurate, the logical-OR operator || works so that it does not check the truth value of its second operand if its first operand makes the entire boolean expression true.)

This statement reads a value for the **char** variable **character_from_keyboard**. This is achieved by first reading a line of text from the keyboard, and then using the **charAt()** method to take the first character of the text line.

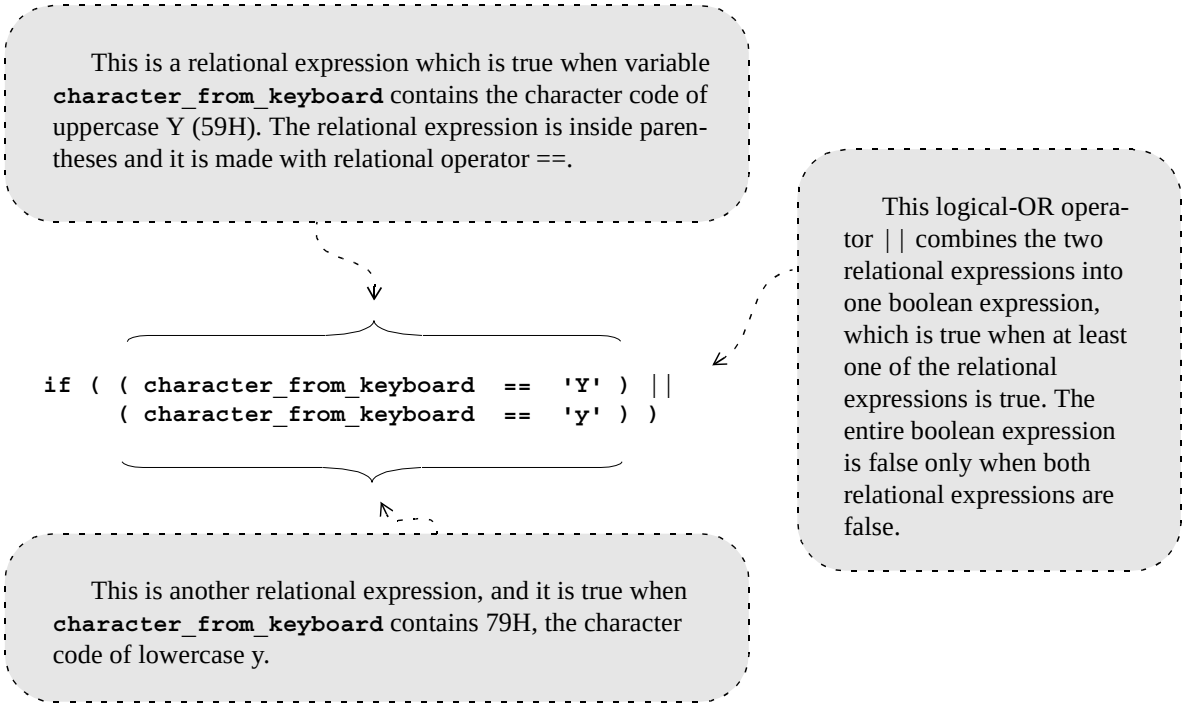
'Y' is a character literal which means 59H, the character code of uppercase letter Y. Single quotes are used to write character literals in Java. Character literals are sometimes called character constants or literal character constants.

```
//  Likejava.java  (c) Kari Laitinen

import java.util.* ;

class Likejava
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      char  character_from_keyboard ;

      System.out.print( "\n Do you like the Java programming language?"
                    +  "\n Please, answer Y or N :  " ) ;

      character_from_keyboard  =  keyboard.nextLine().charAt( 0 ) ;

      if ( ( character_from_keyboard  ==  'Y' ) ||
           ( character_from_keyboard  ==  'y' ) )
      {
         System.out.print( "\n That's nice to hear. \n" ) ;
      }
      else if ( ( character_from_keyboard  ==  'N' ) ||
                ( character_from_keyboard  ==  'n' ) )
      {
         System.out.print( "\n That is not so nice to hear. "
                        +  "\n I hope you change your mind soon.\n" ) ;
      }
      else
      {
         System.out.print( "\n I do not understand \""
                        +  character_from_keyboard   +  "\".\n" ) ;
      }
   }
}
```

This statement will be executed if **character_from_keyboard** contains something other than 'Y', 'y', 'N', or 'n'. Note that if you want to include a double quote character (") among the text to be printed, you must write a backslash \ before it.

This boolean expression is true if **character_from_keyboard** contains either the character code for uppercase N or lowercase n. || is the logical-OR operator which can combine relational expressions.

Likejava.java - 1.+ A program containing an if-else if-else construct.



Likejava.java - 1-1. The first boolean expression in the program.

```
D:\javafiles2>java Likejava

 Do you like the Java programming language?
 Please, answer Y or N :  y

 That's nice to hear.

D:\javafiles2>java Likejava

 Do you like the Java programming language?
 Please, answer Y or N :  z

 I do not understand "z".
```

Likejava.java - X. The program is executed here twice, with inputs y and z.

Character codes are tested in these boolean expressions. Codes that are less than 20H are non-printable characters. Numbers are in the range from 30H to 39H, uppercase letters in the range from 41H to 5AH, and lowercase letters in the range from 61H to 7AH. ' ' means the character code of space (20H), '9' means the character code of digit 9 (39H). && is the logical-AND operator which returns true only if the relational expressions on both sides of && are true.

```
//  Iffing.java

import java.util.* ;

class Iffing
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      char  given_character ;

      System.out.print( "\n Please, type in a character:  " ) ;

      given_character  =  keyboard.nextLine().charAt( 0 ) ;

      if ( given_character  <  ' ' )
      {
         System.out.print( "\n That is an unprintable character \n" ) ;
      }
      else if ( given_character >= '0'  &&  given_character <= '9' )
      {
         System.out.print( "\n You hit the number key "
                        +  given_character  +  ". \n " ) ;
      }
      else if ( given_character >= 'A'  &&  given_character <= 'Z' )
      {
         System.out.print( "\n "  +  given_character
                        +  " is an uppercase letter. \n" ) ;
      }
      else if ( given_character >= 'a'  &&  given_character <= 'z' )
      {
         System.out.print( "\n "  +  given_character
                        +  " is a lowercase letter. \n" ) ;
      }
      else
      {
         System.out.print( "\n "  +  given_character
                        +  " is a special character. \n" ) ;
      }
   }
}
```

Iffing.java - 1. A program that contains a complex if construct.

```
D:\javafiles2>java Iffing

 Please, type in a character:  z

 z is a lowercase letter.
```

Iffing.java - X. The program executed with input z.

Program **Iffing.java** examines the character code of the character that it receives from the keyboard. It can find out whether the character is an unprintable character, a number, an uppercase letter, or a lowercase letter. If the character is none of these, the program assumes that the character is a punctuation character or some other special character. **Iffing.java** has the boolean expression

```
( given_character  >=  '0'   &&  given_character  <=  '9' )
```

to check whether variable **given_character** contains the character code of a number. In this boolean expression the logical-AND operator (&&) is used to combine the two relational expressions. The logical-AND operator returns true only when both of its operands are true. In the above boolean expression, the relational expressions are the operands of the logical-AND operator. If either or both of the relational expressions is false, the entire boolean expression is false. When variable **given_character** contains a value that is greater or equal to the character code of zero (30H) and less than or equal to the character code of nine (39H), the boolean expression above is true.

The boolean expression above could be written, without changing its meaning, with extra parentheses, in the following way

```
((given_character  >=  '0') && (given_character  <=  '9'))
```

but the extra parentheses are not necessary because the operator && has a lower precedence than the relational operators >= and <=. Precedence of operators means the order in which the operators take effect. Operators with higher precedence are applied before operators with lower precedence. In the boolean expression above, operators >= and <= take precedence over operator &&. You can think that the program first evaluates the truth values of the relational expressions

```
given_character  >=  '0'
given_character  <=  '9'
```

and then the calculated truth values are used in a logical-AND operation. In reality, however, the truth value of the second operand is not checked in a logical-AND operation if the first operand of && is false.

Java operators are listed in the order of their precedence in Appendix A. The official precedence of operators can always be overridden with parentheses. For example, the multiplication operator * has a higher precedence than the addition operator +, but with parentheses this precedence can be changed. The following example shows the effect of the use of parentheses:

```
2  *  5  +  4      would be evaluated to 14, but
2  *  ( 5 + 4 )   would be evaluated to 18.
```

The logical operators are listed in Table 6-2 and their behavior is summarized in Table 6-3. The NOT operator ! is said to be an unary operator because it takes only one operand. The NOT operator complements the truth value of its operand expression. Complementing is sometimes called inverting or reversing. Complementing means that either true becomes false or false becomes true. The NOT operator can be used to write complex expressions. For example, the expressions

```
( given_character  <=  '9' )
( ! ( given_character  >  '9' ) )
```

mean the same but the latter expression is more difficult to read and understand. It is better to favor simple expressions. The NOT operator (!) is often useful, but it should not be used unnecessarily.

Table 6-2: The logical operators of Java.[a]

Operator symbol	Operator name	Explanation
`||`	Logical OR	Logical OR returns true if either or both of its operand expressions is true.
`&&`	Logical AND	Logical AND returns true only if both of its operand expressions are true.
`!`	NOT	NOT operator complements the truth value of the expression to the right of the operator symbol !.

a. Java has operators & and | which work in many cases like operators && and ||, respectively. It is, however, better to use operators && and || when you write boolean expressions for `if` constructs and loops. Operators & and | will be discussed in Chapter 16.

Table 6-3: The truth values of logical operations. (a and b can be variables or expressions.)

`a`	`b`	`a || b`	`a && b`	`! a`	`! b`
false	false	false	false	true	true
false	true	true	false	true	false
true	false	true	false	false	true
true	true	true	true	false	false

Exercises related to if constructs

Exercise 6-1. Make a copy of program **Largeint.java**. You might name the new file **Largest4.java**. Modify the new file so that it asks for four integers from the keyboard and finds the largest of the four integers. Having done that, make the program find both the smallest and the largest of the four integers.

Exercise 6-2. Write a program that can convert amounts of currency given in U.S. dollars to Japanese yen and vice versa. Check out the latest currency exchange rates in a newspaper. Your program must first ask whether the user wants to change dollars to yen or yen to dollars. The program must have an `if` construct which selects the right conversion statements. Naturally, if you prefer, you can also use other currency units in your program.

Remember that Appendix C contains advice for programming exercises.

6.2 Making decisions with switch-case constructs

Various kinds of **if** constructs are the principal means for making decisions in Java programs. In addition to **if** constructs, Java provides **switch-case** constructs to make decisions. **switch** and **case** are keywords in Java. You can make all decisions with **if** constructs, but with **switch-case** constructs you can sometimes make the decisions with less writing.

In this section we shall study programs that contain **switch-case** constructs. You need to learn these constructs in order to master Java, and to understand programs you find in the literature. However, because **switch-case** constructs are not used much in this book, you can temporarily skip this section if you are eager to learn more interesting things about Java. You can come back to this section when you see the keywords **switch** and **case** somewhere.

A **switch-case** construct is sometimes convenient when you need to test the value of some variable many times. For example, in program **Likejava.java** the value of variable **character_from_keyboard** is tested altogether four times in two boolean expressions. To demonstrate the use of a **switch-case** construct, the **if-else if-else** construct of **Likejava.java** is replaced with a **switch-case** construct in **Likejavas.java**. Program **Likejavas.java** is thus a rewritten version of **Likejava.java**. This again proves that there can be two programs that behave precisely in the same way although they are written in different ways.

Figure 6-4 shows how **switch-case** constructs usually look like. The structure of a **switch-case** construct is often such that you test the value of an arithmetic expression, and jump to different cases in the construct depending on the value of the tested arithmetic expression. When the program execution goes to some **case** inside the braces of the construct, the execution continues from that point until a **break** statement is encountered. **break**, which is a reserved word and a statement of its own, has the effect that the execution of the program breaks out from the area surrounded by braces. Program **Sentence.java** is an example which further clarifies the role of **break** statements in **switch-case** constructs.

```
switch ( arithmetic expression )
{
case v1:
     Statements which will be executed if the arithmetic expression
     has value v1
        break ;
case v2:
     Statements which will be executed if the arithmetic expression
     has value v2
        break ;
case vn:
     Statements to be executed when the arithmetic expression has
     value vn
        break ;
default:
     Statements which will be executed if none of the cases matched
     the value of the arithmetic expression
        break ;
}
```

Figure 6-4. The structure of a typical switch-case construct.

Often, **switch-case** constructs test the value of a single variable of type **char** or **int**. The variable that is tested is written in parentheses after the keyword **switch**. A single variable is, in fact, a simple arithmetic expression. It is also possible to write a more complex arithmetic expression inside the parentheses. In that case, switching is carried out according to the evaluated value of the arithmetic expression.

```
//  Likejavas.java

import java.util.* ;

class Likejavas
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      char  character_from_keyboard ;

      System.out.print( "\n Do you like the Java programming language?"
                    +  "\n Please, answer Y or N :  " ) ;

      character_from_keyboard  =  keyboard.nextLine().charAt( 0 ) ;

      switch ( character_from_keyboard )
      {
      case  'Y':
      case  'y':
         System.out.print( "\n That's nice to hear. \n" ) ;
         break ;
      case  'N':
      case  'n':
         System.out.print( "\n That is not so nice to hear. "
                       +  "\n I hope you change your mind soon.\n" ) ;
         break ;
      default:
         System.out.print( "\n I do not understand \""
                       +  character_from_keyboard   +  "\".\n" ) ;
         break ;
      }
   }
}
```

A **switch-case** construct is indeed like a switch that you can turn to many positions. The cases inside braces are the possible positions where the program execution can go in the switching operation. The execution of the program jumps from the keyword **switch** into this position when **character_from_keyboard** contains either 'N' or 'n'. The **break** statements cause the program execution to jump outside the **switch-case** construct.

The case marked with keyword **default** is the place where the program execution jumps to if none of the other cases match the contents of **character_from_keyboard**.

Likejavas.java - 1. Program Likejava.java rewritten using a switch-case construct.

```
D:\javafiles2>java Likejavas

 Do you like the Java programming language?
 Please, answer Y or N :  5

 I do not understand "5".
```

Likejavas.java - X. The program is executed here with input 5.

Exercises related to boolean expressions

Exercise 6-3. Let's suppose that **`first_variable`** has value 5, **`second_variable`** has value 8, and **`third_variable`** contains value 14. Write T or F after the following boolean expressions depending on whether they are true or false!

```
( first_variable  <  second_variable )
( third_variable  <  first_variable )
( ! ( first_variable  <  second_variable ) )
( third_variable  >  first_variable )
( ( first_variable + second_variable )  <  third_variable )
( ( first_variable + second_variable )  <= third_variable )
( ( third_variable - second_variable )  >  first_variable )
( first_variable  ==  0 )
( ! ( first_variable == 0 ) )
( first_variable  >  0  ||  second_variable  <  0 )
( first_variable  == 8  ||  second_variable  == 5 )
( first_variable  <  second_variable  &&  third_variable >=  14 )
( first_variable  == 5  &&  second_variable  >  8 )
```

Exercise 6-4. Write the following complex boolean expressions in a simpler form

```
( ! ( some_variable  == 8 ) )
( ( some_variable - 10 )  >  0 )
( some_variable  <  88  ||  some_variable  ==  88 )
( ( some_variable - other_variable )  <  0 )
( ! ( ! ( some_variable  <  99 ) ) )
```

This program, which is a somewhat illogical program, clarifies the role of the **break** statements in **switch-case** constructs. As a general rule, programs should not be written like this. There should always be a **break** statement after the statements of each **case** in a **switch-case** construct.

This statement reads a value for the **char** variable **character_from_keyboard**. The input from the keyboard is processed so that first the method **nextLine()** reads a line of text, then the method **toUpperCase()** converts all characters of the given text line to uppercase letters, and finally method **charAt()** takes the first character of the text line. As the text characters are converted to uppercase letters, the user of the program does not need to worry about the case of the letters.

```
//  Sentence.java

import java.util.* ;

class Sentence
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      char  character_from_keyboard ;

      System.out.print(
              "\n Type in L, M, or S, depending on whether you want"
           +  "\n a long, medium, or short sentence displayed:  "  ) ;

      character_from_keyboard  =
                   keyboard.nextLine().toUpperCase().charAt( 0 ) ;

      System.out.print( "\n This is a" ) ;

      switch ( character_from_keyboard )
      {
      case  'L':
         System.out.print( " switch statement in a \n" ) ;
      case  'M':
         System.out.print( " program in a" ) ;
      case  'S':
         System.out.print( " book that teaches Java programming." ) ;
      default:
        System.out.print( "\n I hope that this is an interesting book.\n");
      }
   }
}
```

Because there are no **break** statements in this **switch-case** construct, all statements following a certain **case** will be executed. For example, when the user of the program types in the letter L, all statements inside the **switch-case** construct will be executed because **case 'L':** happens to be the first case.

Sentence.java - 1. Using a switch-case construct with no break statements.

```
D:\javafiles2>java Sentence

 Type in L, M, or S, depending on whether you want
 a long, medium, or short sentence displayed:  s

 This is a book that teaches Java programming.
 I hope that this is an interesting book.

D:\javafiles2>java Sentence

 Type in L, M, or S, depending on whether you want
 a long, medium, or short sentence displayed:  L

 This is a switch statement in a
 program in a book that teaches Java programming.
 I hope that this is an interesting book.
```

Sentence.java - X. The program is executed here twice, with inputs s and L.

6.3 while loops enable repetition

At this point you should have realized that computers are actually quite stupid machines. To make them do something, you have to very carefully and precisely describe that thing in a source program. But what computers lack in intelligence they gain in speed. Once you have written a program that works correctly, computers can execute the program extremely fast–that is, millions of machine instructions per second. And you can run the same program equally fast, as many times as you like, in many different computers if you choose. Since computers can do things so fast, they can be made to be very effective by making them repeat things.

Loops are program constructs with which we can make computers repeat things over and over. As computers are so fast, we can accomplish many things simply by making a computer repeat a few simple statements. There are several different loop structures in Java, but the **`while`** loop can be considered the basic loop. The structure of **`while`** loops is described in Figure 6-5.

Programs that we have studied so far have been such that they are executed from the beginning to end, statement by statement. When there are **`if`** constructs in a program, they have the effect that some statements may be left unexecuted, but programs with **`if`** constructs are also executed from the beginning to end. When there are loops in a program, the execution of statements is not always from the beginning to end. With a **`while`** loop, for example, there is a possibility to jump backwards in a program, and execute the internal statements of the loop over and over. The statements of a program are always executed in the order they are written in the source program, but a loop allows the program execution to go through the same statements many times. When a **`while`** loop is encountered in a program during its execution, we can imagine that the computer follows these three steps:

Step 1. Check the truth value of the boolean expression given in parentheses.

Step 2. If the boolean expression is true, execute the internal statements once and go back to Step 1.

Step 3. If the boolean expression is false (not true), continue by executing the statements that follow the **`while`** loop in the program.

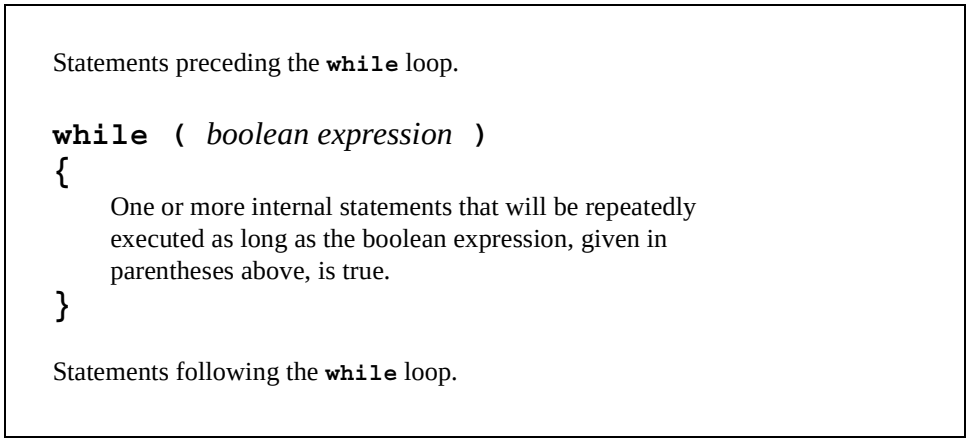
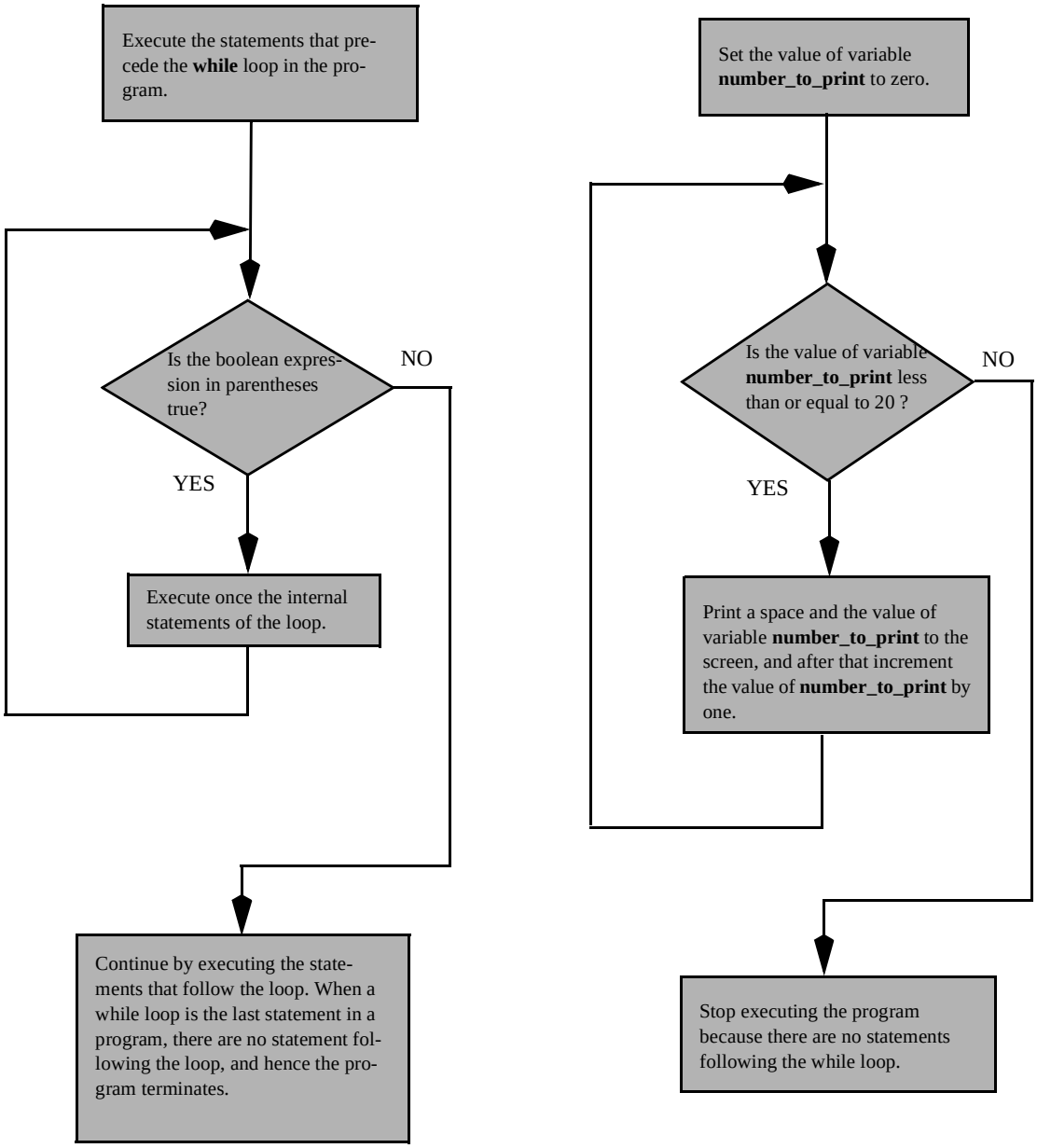


Figure 6-5. The structure of while loops.

Describing while loops with flowcharts

A flowchart is a traditional way to describe the operation of a program. A flowchart shows graphically how the program control flows within a program. Flowcharts are particularly useful to explain how loops operate. Below on the left you find a flowchart that describes the general operation principle of a `while` loop. The flowchart on the right shows how the `while` loop in program **While20.java** operates. The arrows in the flowcharts represent movements from one activity to another in the program. The rectangles describe activities, and the diamond shapes describe conditions.



Program **While20.java** is an example where a **`while`** loop is used to print the numbers from zero to 20 to the screen. The two statements inside the **`while`** loop will be repeated 21 times when the program is run. The essential idea in the **`while`** loop is that the internal statements of the loop modify a variable that affects the truth value of the boolean expression. In **While20.java**, the value of variable **`number_to_print`** grows inside the loop. The same variable is tested in the boolean expression, and ultimately the value of **`number_to_print`** is so big that the boolean expression becomes false, and the loop terminates. When the boolean expression of a **`while`** loop is not true any more, the program execution continues from the statement that follows the **`while`** loop in the program. In **While20.java**, the whole program terminates when the **`while`** loop terminates because there are no statements following the **`while`** loop.

Program **Whilesum.java** is another example of the use of a **`while`** loop. The program reads integers from the keyboard and maintains the sum of the integers read. The sum is displayed each time the internal statements of the loop are executed. The loop is repeated as many times as the user types in an integer. The execution of the loop, and the entire program, is terminated when the user enters a zero from the keyboard.

The internal statements of a **`while`** loop are executed zero times if the boolean expression is false at the beginning. Rarely executing **`while`** loops are sometimes needed in computer programs, but **`while`** loops that are never entered are programming mistakes. It is often too easy to make a mistake such that the boolean expression of a **`while`** loop is never true. For example, in **Whilesum.java**, if we initialized variable **`integer_from_keyboard`** to zero, the internal statements of the **`while`** loop would be never entered.

Another common programming mistake is to write a **`while`** loop that never terminates. If the execution of the internal statements does not affect the truth value of the boolean expression, the loop is most likely a never-ending, infinite (endless) loop. In older personal computers the only possibility to terminate an endless loop was to switch off electricity from the computer. This ultimate loop-termination act may still be sometimes needed, but it is better first to try to close the window where the program is executing, or press Control-C (Ctrl and C keys simultaneously on the keyboard). We get an example of an endless loop by writing the **`while`** loop of program **While20.java** in the following way:

```
while ( number_to_print  <=  20 )
{
   System.out.print( " "  +  number_to_print ) ;
}
```

The loop above is an endless loop, because variable **`number_to_print`** is not incremented inside the loop, and the boolean expression stays true forever. If the loop above is inserted in program **While20.java** into the place of the existing **`while`** loop, the program would keep printing the number zero forever.

It is possible to declare a variable and assign a value to it in a single statement. This line means the same as

```
int number_to_print ;
number_to_print  =  0 ;
```

When a variable is assigned a value at the same time when it is declared, we say that the variable is an initialized variable.

```
//  While20.java

class While20
{
   public static void main( String[] not_in_use )
   {
      int  number_to_print  =  0 ;

      System.out.print( "\n Numbers from 0 to 20 are the following:\n\n ") ;

      while ( number_to_print  <= 20 )
      {
         System.out.print( " "  +  number_to_print ) ;
         number_to_print  ++  ;
      }
   }
}
```

++ is called the increment operator. In this case the operator increments the value of **number_to_print** by one. This line means the same as

```
number_to_print = number_to_print + 1 ;
```

The two statements inside braces after the boolean expression will be repeatedly executed as long as the boolean expression is true. As the value of **number_to_print** is initially zero, and it is incremented by one every time the loop is executed, the loop will terminate after 21 repetitions.

The above program could be constructed without a loop by writing the two statements 21 times in the program:

```
System.out.print( " "  +  number_to_print ) ;
number_to_print  ++  ;
System.out.print( " "  +  number_to_print ) ;
number_to_print  ++  ;
System.out.print( " "  +  number_to_print ) ;
number_to_print  ++  ;
  ... etc. etc.
```

It is easier, though, to write the program by using a loop.

While20.java - 1. A program containing a simple while loop.

```
D:\javafiles2>java While20

 Numbers from 0 to 20 are the following:

  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

While20.java - X. The output from the program is always the same.

These variables are assigned initial values at the same time they are declared. **integer_from_keyboard** must be initialized with a non-zero value because otherwise the boolean expression of the **while** loop would not be true at the beginning. **sum_of_integers** must be zero at the beginning because no integers have been read from the keyboard so far.

This boolean expression is true as long as **integer_from_keyboard** contains something other than a zero. != is the "not equal" operator of Java

```
//  Whilesum.java  (c) Kari Laitinen

import java.util.* ;

class Whilesum
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      int  integer_from_keyboard   =  -1 ;
      int  sum_of_integers         =   0 ;

      System.out.print( "\n This program calculates the sum of the integers"
                     + "\n you type in from the keyboard. By entering a"
                     + "\n zero you can terminate the program. \n\n" ) ;

      while ( integer_from_keyboard  !=  0 )
      {
         System.out.printf( "  Current sum: %8d    Enter an integer: ",
                         +  sum_of_integers  ) ;

         integer_from_keyboard  =  keyboard.nextInt() ;

         sum_of_integers  =  sum_of_integers + integer_from_keyboard ;
      }
   }
}
```

Variable **integer_from_keyboard** gets a new value each time the internal statements of the loop are executed. Immediately after the integer is read from the keyboard, it is added to the sum of the integers.

The value of variable **sum_of_integers** is printed right-justified into a printing field that is 8 character positions wide. This is achieved by using the format specifier **%8d** with the **printf()** method. When the internal statements of the loop are executed for the first time, **sum_of_integers** is zero.

Whilesum.java - 1. A program to calculate the sum of integers in a while loop.

```
D:\javafiles2>java Whilesum

 This program calculates the sum of the integers
 you type in from the keyboard. By entering a
 zero you can terminate the program.

  Current sum:        0    Enter an integer: 5
  Current sum:        5    Enter an integer: 16
  Current sum:       21    Enter an integer: 107
  Current sum:      128    Enter an integer: 1008
  Current sum:     1136    Enter an integer: 9999
  Current sum:    11135    Enter an integer: 0
```

Whilesum.java - X. The program calculates the sum of six integers here.

while loops, and also other loops, are such that we often use a particular integer variable to control the correct termination of the loop. The technique works by having a variable as part of the boolean expression of the loop, and this same variable is incremented or decremented inside the loop. To increment and decrement the values of variables, Java provides two operators ++ and --. Operator ++ is the increment operator which increments the value of a variable by one. Operator -- is the decrement operator which subtracts one from the value of the variable. Incrementing the value of a variable by one is the same as assigning the variable a value which is one larger than its current value. Thus the meaning of the following two statements is the same:

```
some_variable  ++  ;
some_variable  =  some_variable  +  1 ;
```

Also the meaning of the following two statements is the same:

```
some_variable  -- ;
some_variable  =  some_variable  -  1 ;
```

Increment and decrement operators are useful because incrementing and decrementing operations are so common inside loops, and these operators allow us to write things down in a concise way. When you use an increment or decrement operator in a program, you may not write any spaces between the two plus or minus signs. If you write, for example, + +, the compiler interprets these separate plus signs as two adjacent addition operators, and most likely displays an error message.

Assignment statements and increment/decrement statements, such as the ones above, are usually short, occupying only one line in a program. They look very different from loops or **if** constructs. However, it is important to realize also that **while** loops, and the other loops that will be introduced in the following sections, are statements. Various kinds of **if** constructs are also statements. Because loops and **if** constructs are usually long and consist of many lines of source program, they do not look very similar to the shorter statements. For this reason, they are just called loops and constructs in this book, but you must remember that loops and **if** constructs are statements. A loop can have another loop or an **if** construct as an internal statement.

6.4 for loops repeat a known number of times

Program **While20.java** uses a `while` loop to print numbers from 0 to 20. The internal statements of the loop are repeated 21 times. The termination of the loop depends on the value of variable `number_to_print`. Program **For20.java** is a rewritten version of program **While20.java**. A `for` loop is used in **For20.java** instead of a `while` loop. Although they are written by using different looping mechanisms, programs **For20.java** and **While20.java** perform in exactly the same way.

`for` loops are convenient when we want to repeat something a certain number of times in a program. Typically, `for` loops are controlled by a single integer variable (`number_to_print` in **For20.java**) which is either incremented or decremented each time the internal statements of the loop have been executed. The general structure of a typical `for` loop is described in Figure 6-6. `for` loops are not that much different from `while` loops. The termination of both loops is controlled by a single boolean expression. When the boolean expression is or becomes false, the execution of the program continues from the statement that follows the loop. The essential difference between a `while` loop and a `for` loop is that a `for` loop has three "things" inside parentheses ( ) after keyword `for`, whereas a `while` loop has only one "thing", the boolean expression, inside parentheses after keyword `while`. The assignment statement that is the first thing in parentheses in Figure 6-6 will be executed before anything else takes place, and that statement is executed only once. The third thing in parentheses, the increment or decrement statement, is executed always after the internal statements of the loop have been executed.

Everything that can be done with a `for` loop in a program can also be done with a `while` loop. Figure 6-7 shows how a `for` loop can be converted into a `while` loop. The assignment statement, that is the first item inside parentheses of the `for` loop, can be written as a statement that precedes the `while` loop. The increment or decrement statement, that is the last item inside parentheses of the `for` loop, can be added as the last statement to the body of the `while` loop. As the increment or decrement statement is executed equally as many times as the internal statements of a `for` loop, it is logical to include it in the internal statements of the corresponding `while` loop.

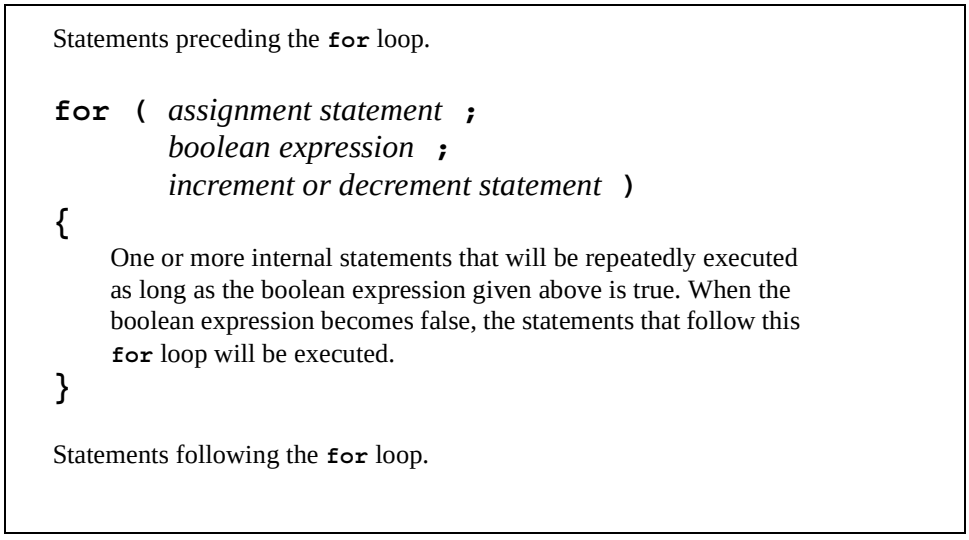


Figure 6-6. Typical structure of a for loop.

Inside the parentheses after the keyword **for**, **for** loops have three "things" separated with two semicolons. In this loop

- the assignment statement **number_to_print = 0** will be executed before the program actually starts looping,
- the boolean expression **number_to_print <= 20** decides when the loop terminates, and
- the increment statement **number_to_print ++** will be executed each time after the internal statement of the loop has been executed.

```
//  For20.java  (c) Kari Laitinen

class For20
{
   public static void main( String[] not_in_use )
   {
      int  number_to_print ;

      System.out.print( "\n Numbers from 0 to 20 are the following:\n\n ") ;

      for ( number_to_print  =  0  ;
            number_to_print  <= 20 ;
            number_to_print  ++  )
      {
         System.out.print( " "  +  number_to_print ) ;
      }
   }
}
```

In the same way as in the case of **while** loops, the internal statements of **for** loops are written inside braces. The internal statements of a loop can also be called with the term "body of the loop".

This statement is the only statement inside the loop. The statement will be executed 21 times. When the value of **number_to_print** is 20, it will be incremented to 21, resulting in that the boolean expression

number_to_print <= 20

is not true any more, and the loop terminates.

For20.java - 1. Program While20.java implemented with a for loop.

```
D:\javafiles2>java For20

 Numbers from 0 to 20 are the following:

  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

For20.java - X. The program produces the same output as program While20.java

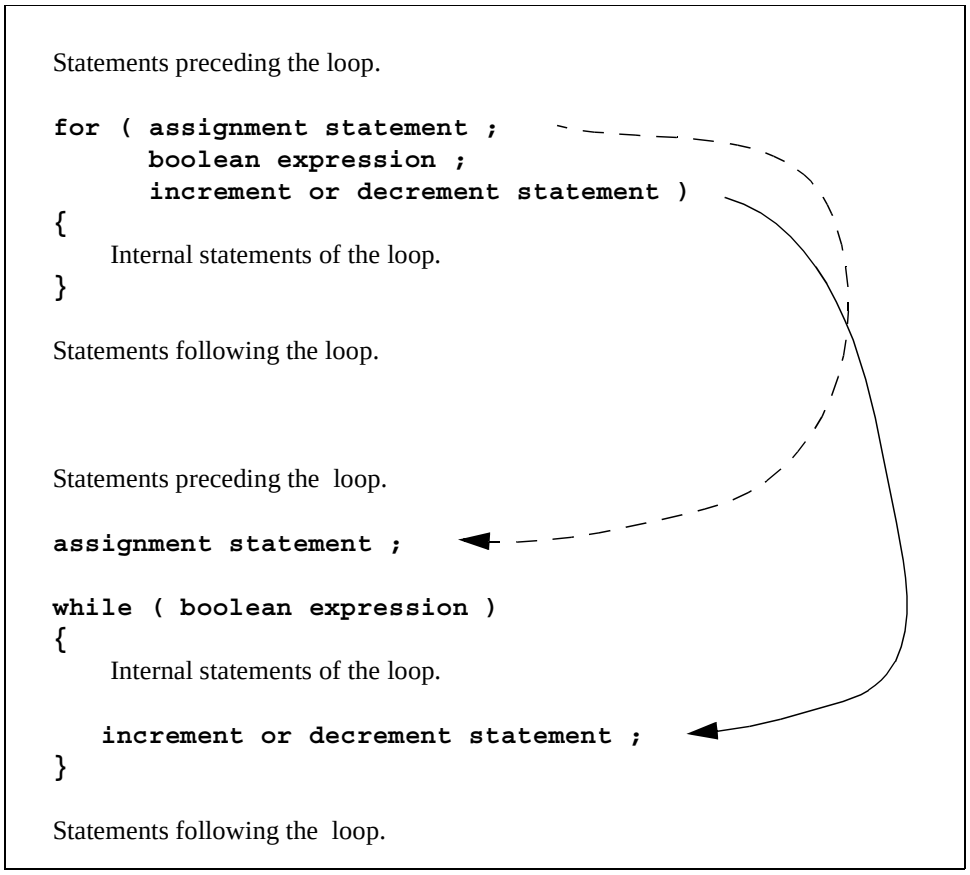


Figure 6-7. Converting a for loop into a while loop

Loops and `if` constructs can be mixed in a program. A loop can contain an `if` construct, and a loop can be written inside an `if` construct. Program **Forcodes.java** shows how an `if` construct can be one of the internal statements of a `for` loop. When the internal statements of the `for` loop of **Forcodes.java** are repeated, the internal statements of the `if` construct are executed when the boolean expression of the `if` construct is true.

Forcodes.java prints characters and their character codes in the range from 20H to 7FH (from 32 to 127 in decimal). The program must repeat the internal statements of the loop 96 times to perform the entire printing operation. Because we know beforehand how many times the loop must be repeated, a `for` loop is convenient in this program. To print the characters and character codes from 20H to 7FH, the loop control variable named `numerical_code` is set to value 0x20 at the beginning and the loop terminates when `numerical_code` reaches value 0x80. A `for` loop is typically such that its loop control variable is given a certain initial value, and its value is incremented or decremented until it reaches a certain terminal value. In program **Forcodes.java**, the loop control variable `numerical_code` passes through all values from 20H to 7FH while the loop is being executed.

Hexadecimal literals 0x20 and 0x80 are used in **Forcodes.java**. The prefix 0x is needed when a programmer wants to write a numerical literal in hexadecimal form in Java. When the compiler recognizes the prefix 0x, it knows that it is a hexadecimal number. Without a prefix, the compiler assumes numerical literals to be in the normal decimal form. It is often convenient to think of character codes in hexadecimal form. However, if you found hexadecimal literals difficult in **Forcodes.java**, you could replace 0x20 with 32 and 0x80 with 128 without making any functional changes in the program.

As program **Forcodes.java** must print 96 different characters and their character codes and also insert spaces to separate the characters and codes, it is impossible to print everything on a single line on the computer's screen. For this reason, the program brings a new line into use after it has printed 8 characters and codes on the current line. The program uses the variable **number_of_codes_on_this_line** to count how many character codes it has processed. The **if** construct in the body of the **for** loop monitors the value of this variable. When **number_of_codes_on_this_line** reaches the value 8, a newline character is printed and the value of the variable is made zero again.

In the case of each numerical code **Forcodes.java** first prints the character and then the numerical code in hexadecimal form. For example, when the value of variable **numerical_code** is 54H, the program prints "T 54 ". The program does not use letter H to indicate hexadecimal numbers. To convert a numerical code into a character, the program copies the numerical code into a variable of type **char** in the statement

```
char character_to_print  = (char) numerical_code ;
```

which both declares the variable and copies a value to it. The marking **(char)** in the above statement is an explicit type conversion that converts the value of **numerical_code** to type **char** before the assignment takes place. Without the explicit type conversion, the above assignment is not possible. After the above statement has been executed, both variables, **character_to_print** and **numerical_code**, contain the same numerical value. But the contents of variable **character_to_print** will be printed as a character because it is of type **char**. Here you must remember that all variables in a computer's main memory contain nothing but binary numbers, zeroes and ones. But in a program which uses the variables in a computer's memory, the binary information stored in the variables can be interpreted in different ways, depending on the type of the variable. In **Forcodes.java**, the numerical information stored in variable **numerical_code** is considered to be numerical information, but when the same information is stored in the variable **character_to_print**, it is treated as a character symbol.

Exercises related to while loops

Exercise 6-5. Which numbers would be printed to the screen if the lines

```
int  growing_number    =  1 ;
int  shrinking_number  =  20 ;

while ( growing_number  <  shrinking_number )
{
   System.out.print(  " "  +  growing_number
                    +  " "  +  shrinking_number ) ;

   growing_number  =  growing_number  +  2 ;
   shrinking_number  =  shrinking_number  -  3 ;
}
```

were executed on a computer? How would the output of the above **while** loop change if the internal statements of the loop were put in an opposite order, i.e., if the output statement came after the assignment statements?

Exercise 6-6. Make a copy of program **While20.java**, and name the new file **Whileodd.java**. Modify the new program so that it prints only the odd numbers in the range from 0 to 20.

Here a variable of type **char** is declared inside the **for** loop, and the value of **numerical_code** is copied into the variable. Variable **character_to_print** is printed as a character because it is of type **char**. Variable **numerical_code** is printed as numerical digits because it is of type **int**.

By using the **printf()** method and the format specifier **%x**, it is possible to print the value of an **int** variable in hexadecimal form. The current value of **numerical_code** replaces the format specifier **%x** in the string **"%x "**. In practice this means that a space character is printed after each hexadecimal code.

```
//  Forcodes.java  (c) Kari Laitinen

class Forcodes
{
   public static void main( String[] not_in_use )
   {
      int   number_of_codes_on_this_line  =  0 ;

      System.out.print( "\n The visible characters with codes from 20"
                  +  "\n to 7F (hexadecimal) are the following:\n\n ");

      for ( int numerical_code  =  0x20 ;
                numerical_code  <  0x80 ;
                numerical_code  ++ )
      {
         char  character_to_print  =  (char) numerical_code  ;

         System.out.print( character_to_print  +  " " ) ;
         System.out.printf( "%x ", numerical_code ) ;

         number_of_codes_on_this_line  ++  ;

         if (  number_of_codes_on_this_line  ==  8  )
         {
            System.out.print( "\n " ) ;
            number_of_codes_on_this_line   =  0 ;
         }
      }
   }
}
```

We say that a program prints a newline, when a new empty line is started on the screen. A newline can be printed simply by outputting character \n which means actually a character which has character code 0AH (10 decimal).

This **if** construct ensures that the program prints a newline after eight characters and their character codes have been printed. The boolean expression of the **if** construct becomes true in every 8th repetition of the loop. The program prints altogether 96 characters and character codes on 12 lines.

Forcodes.java - 1.+ A program that prints a character code table.

It is possible to declare and initialize a variable inside the parentheses of a **for** loop. Because the initial value of the variable **numerical_code** is 20H (32 decimal), a space is the first character to be printed. 20H is the character code of the space character. By adding the prefix 0x before the actual number, you can define a hexadecimal literal constant in Java. 0x20 means the same as 32.

The internal statements of the loop will be repeatedly executed as long as the value of **numerical_code** is less than 80H (128 decimal).

Variable **numerical_code** is incremented by one every time after the internal statements of the loop have been executed once.

```
for ( int numerical_code  =  0x20 ;
          numerical_code  <  0x80 ;
          numerical_code  ++ )
{
   char  character_to_print  =  (char) numerical_code  ;

   System.out.print( character_to_print  +  " " ) ;
   System.out.printf( "%x ", numerical_code ) ;

   number_of_codes_on_this_line  ++  ;

   if (  number_of_codes_on_this_line  ==  8  )
   {
      System.out.print( "\n " ) ;
      number_of_codes_on_this_line   =  0 ;
   }
}
```

Forcodes.java - 1 - 1. The for loop which prints the characters and character codes.

```
D:\javafiles2>java Forcodes

 The visible characters with codes from 20
 to 7F (hexadecimal) are the following:

   20 ! 21 " 22 # 23 $ 24 % 25 & 26 ' 27
 ( 28 ) 29 * 2a + 2b , 2c - 2d . 2e / 2f
 0 30 1 31 2 32 3 33 4 34 5 35 6 36 7 37
 8 38 9 39 : 3a ; 3b < 3c = 3d > 3e ? 3f
 @ 40 A 41 B 42 C 43 D 44 E 45 F 46 G 47
 H 48 I 49 J 4a K 4b L 4c M 4d N 4e O 4f
 P 50 Q 51 R 52 S 53 T 54 U 55 V 56 W 57
 X 58 Y 59 Z 5a [ 5b \ 5c ] 5d ^ 5e _ 5f
 ` 60 a 61 b 62 c 63 d 64 e 65 f 66 g 67
 h 68 i 69 j 6a k 6b l 6c m 6d n 6e o 6f
 p 70 q 71 r 72 s 73 t 74 u 75 v 76 w 77
 x 78 y 79 z 7a { 7b | 7c } 7d ~ 7e ¦ 7f
```

Forcodes.java - X. The 96 characters and character codes printed by the program.

6.5 do-while loops execute at least once

Both **while** loops and **for** loops have a boolean expression which decides whether the internal statements of the loop are executed or not. The boolean expression is checked first, and the internal statements are executed afterwards if the boolean expression was true. The statements inside **while** and **for** loops are not executed at all if the boolean expression is false at the beginning. Sometimes it is necessary that a loop is executed zero times, but in some other cases we need loops to execute their body, the internal statements, at least once. For such situations, Java provides a third possibility to construct a loop. These loops are called **do-while** loops.

Figure 6-8 shows the basic structure of a **do-while** loop. The essential difference between **while** loops and **do-while** loops, is that in **while** loops the value of the boolean expression is evaluated at the beginning, whereas in **do-while** loops the boolean expression is evaluated after the internal statements have been executed. For this reason, the internal statements of a **do-while** loop are executed at least once.

Program **Meanvalue.java** calculates the mean value of the integers read from the keyboard. It uses a **do-while** loop to read the integers and simultaneously calculate the sum of the integers read. The boolean expression of the **do-while** loop causes it to terminate when the user types in a zero from the keyboard. After the **do-while** loop the program calculates the mean value as it knows the sum of the integers and how many integers were entered from the keyboard. Note that an **if** construct is used in the program **Meanvalue.java** to ensure that there were non-zero number of integers entered from the keyboard. No mean value can be calculated if no integers other than the zero were entered.

In program **Meanvalue.java** the following statement calculates the mean value:

```
mean_value  =  (float) sum_of_integers /
               (float) number_of_integers_given ;
```

The term (**float**) in the assignment statement above means that the **int** variables **sum_of_integers** and **number_of_integers_given** are converted to type **float** before division. This kind of conversion is called explicit type conversion. In the above case, type conversion is necessary to get accurate division results. You can make these kinds of type conversions for any variable type by writing the destination type in parentheses before the variable name. Explicit type conversion has a local effect to the type of a variable. The converted value is used only in that place of a program where (*some type*) is written. For example in the statement above, the value of **int** variable **sum_of_integers** is treated as a **float** value, but the variable still remains as an **int** variable, and it would be treated as such if it were used later in the program.

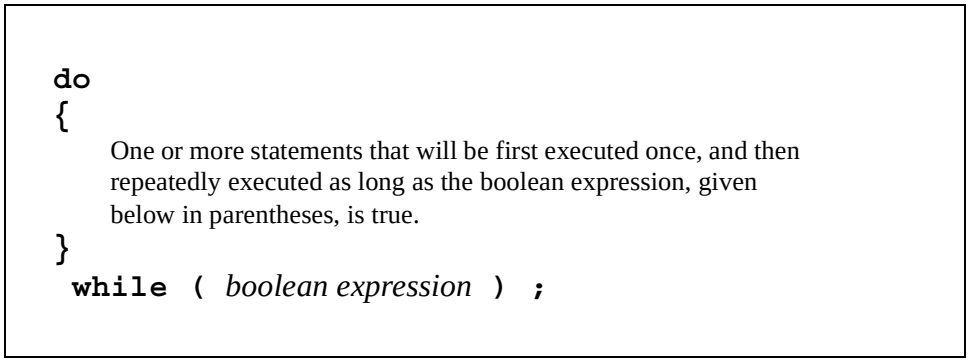


Figure 6-8. The structure of do-while loops.

This **do-while** loop reads the integers and calculates their sum. **do** is a reserved keyword of Java. The internal statements of a **do-while** loop, which are always executed at least once, are given inside braces immediately after the keyword **do**.

-1 is the initial value of the variable that counts how many integers have been entered from the keyboard. This way the last integer, a zero, is not calculated in the sum of the integers.

```
//  Meanvalue.java  (c) Kari Laitinen

import java.util.* ;

class Meanvalue
{
   public static void main( String[] not_in_use )
   {
      Scanner keyboard = new Scanner( System.in ) ;

      int   integer_from_keyboard     =  0 ;
      int   number_of_integers_given  = -1 ;
      float mean_value                =  0 ;
      int   sum_of_integers           =  0 ;

      System.out.print( "\n This program calculates the mean value of"
                     +  "\n the integers you enter from the keyboard."
                     +  "\n Please, start entering numbers. The program"
                     +  "\n stops when you enter a zero. \n\n" ) ;

      do
      {
         System.out.print( "   Enter an integer: " ) ;
         integer_from_keyboard  =  keyboard.nextInt();

         number_of_integers_given  ++  ;
         sum_of_integers  =  sum_of_integers + integer_from_keyboard ;
      }
        while  ( integer_from_keyboard  !=  0 ) ;

      if ( number_of_integers_given  >  0 )
      {
         mean_value  =  (float) sum_of_integers /
                        (float) number_of_integers_given ;
      }

      System.out.print( "\n The mean value is: " + mean_value + " \n" ) ;
   }
}
```

This program calculates the mean value only if some numbers were actually entered from the keyboard. Without this **if** construct, the program could carry out a division by zero which might result in serious problems when the program is executed.

There must be a semicolon to terminate a **do-while** loop. The boolean expression is always evaluated after the internal statements of the loop have been executed once. Here, the boolean expression is constructed by using operator !=, "not equal".

Meanvalue.java - 1. A program to calculate the mean value of a set of integers.

```
D:\javafiles2>java Meanvalue

 This program calculates the mean value of
 the integers you enter from the keyboard.
 Please, start entering numbers. The program
 stops when you enter a zero.

   Enter an integer: 222
   Enter an integer: 333
   Enter an integer: 444
   Enter an integer: 555
   Enter an integer: 0

 The mean value is: 388.5
```

Meanvalue.java - X. The program calculates here the mean value of four integers.

Exercises with loops

Exercise 6-7. Make a copy of program **Whilesum.java**, and modify the new file so that the program calculates how many integers the user has entered from the keyboard. The program must stop reading in new integers when the user has entered 10 integers.

Exercise 6-8. Make a copy of program **Meanvalue.java**, and modify the new file so that the program prints the current mean value each time a new integer has been entered from the keyboard.

Exercise 6-9. Write a program that prints a conversion table from miles to kilometers or from kilometers to miles. The program must first ask what kind of conversion table the user wants. After having asked this, you need an `if` construct in the program. The program must use either a `for` or a `while` loop to print the conversion table. The program must print at least 15 conversion lines, for example, in the following way:

```
miles      kilometers

10         16.09
20         32.19
30         48.28
.          .
.          .
.          .
140        225.30
150        241.40
```

Exercise 6-10. In Chapter 5 there is an exercise which explains how to convert degrees Celsius to degrees Fahrenheit. Write a program that prints a Celsius to Fahrenheit conversion table. Use a `for` loop in your program.

Exercise 6-11. Write a program that can display the character code of any character that is entered from the keyboard. The program should read characters in a loop and print their character codes. The character code must be printed both in hexadecimal and decimal form. You can use program **Forcodes.java** as an example, but it is better to use a `while` loop in this kind of a program. The program should stop asking new characters when the user enters a special character like % or &.