Chapter 2

Overview of the Java Programming Language

This chapter provides a general overview of the key features of the Java programming language. It assumes that you have already been exposed to computer programming and that you know how to write programs in a programming language such as C or Visual Basic®. It does not matter which programming language you know; however, you should be familiar with concepts such as data types, variables, arrays, control structures, and performing input and output.

The chapter starts by explaining how to create, compile, and execute a Java program from within an integrated development environment. It then presents a simple *Hello World* program and gives the steps necessary to create an executable version of this program on your computer. The chapter goes on to describe features such as data types, variables, control structures, and arrays in Java. Numerous examples are given throughout the chapter to reinforce the descriptions. The chapter also contains several exercises carefully designed to get you up and running with Java as quickly as possible. Programming hints and solutions for these exercises can be downloaded from the book Web site.

2.1 Java Development Environment

In order to compile and run a Java program, it is necessary to download the Java compiler and run-time environment. Editing and compiling a Java program can be simplified by using one of the integrated development environments (IDEs) available for Java. These include *JCreator*, *Eclipse*, and *NetBeans*. These IDEs provide facilities for editing, compiling, and running a Java program from within a single environment.

The book Web site provides instructions for installing the Java compiler and run-time environment. It also provides instructions for downloading and installing the three IDEs mentioned above. At the book Web site, you can also get detailed instructions for editing, compiling, and running a Java program in each of the IDEs. The remainder of this chapter assumes that you have installed the Java compiler and run-time environment and that you know how to edit, compile, and run a Java program.

2.2 Hello World

Consider the following Java code:

```
public class HelloWorld
{
   public static void main(String[] args) {
      System.out.println ("Hello World!");
      System.out.println ("This is my first Java program.");
      System.out.println
         ("Now I am ready for object-oriented programming!");
   }
}
```

The above code comprises a complete Java program[1]. You should type the program in an IDE and save it as `HelloWorld.java`. Note that Java is case sensitive so `HelloWorld.java` is different from `helloworld.java`.

Classes are the building blocks with which all Java programs are built. They are discussed in detail in Chapter 3. Almost everything in a Java program must be inside a class. The name of the class must follow the keyword `class`. In the code above, the name of the class is `HelloWorld`. A class name can have any combination of letters and digits (and a few special characters). However, it must begin with a letter.

The name of the file containing the source code must be the same as the name of the class with the word `.java` appended. For example, the `HelloWorld` class must be saved in the file `HelloWorld.java`.

A Java program consists of one or more classes. One of the classes in the program must contain a `main()` method[2]. The `main()` method is automatically called when the program is executed and serves as the entry point into the program.

[1] The term *Java program* is sometimes used in this book to refer to what is formally called a *Java application*.
[2] Classes and methods are discussed in detail in Chapter 3. For the time being, consider a method as a programming feature that is similar to a function.

Braces (curly brackets) are used to delineate the parts or blocks in a Java program just like in C or C++. They are similar to the `begin/end` pairs in Pascal and the `Sub/End Sub` pairs in Visual Basic®.

When the `HelloWorld` program is compiled, a file `HelloWorld.class` is produced if compilation is successful. This is the "executable" version of the source code[3]. If compilation is not successful, the compilation error/s will be displayed on one of the screens of the IDE. These errors must be corrected until a successful compilation is obtained.

When the `HelloWorld` program is executed, the following output is generated:

```
Hello World!
This is my first Java program.
Now I am ready for object-oriented programming!
```

The remainder of this chapter will explore features in Java that you are likely to have encountered in some other programming language. Of course, the object-oriented features of Java are covered throughout the book.

2.3 Comments

Comments in Java, like comments in most programming languages, are ignored by the compiler in producing the executable code. The most common way to write a comment is to use // for a comment that will span only one line:

```
System.out.println ("Hi There!");    //this is a one line comment
```

When longer comments are needed, it is more convenient to use the /* and */ symbols to block off the comment. For example,

```
/*   --------------------------------------------
   This is a longer comment that spans more than
   one line.
   ---------------------------------------------
*/
```

There is a third kind of comment that can be used to generate source code documentation which is formatted for viewing on a Web browser. This type of

[3] The `.class` file is not directly executable on a computer like the `.exe` file generated by a C compiler. It must be interpreted by the Java run-time system.

comment starts with the /** symbol and ends with the */ symbol. More information on this kind of comment is available at the book Web site.

2.4 Primitive Data Types

Java is a *strongly typed language*. This means that every variable must have a declared type. There are eight *primitive* types in Java. Six of them are number types, one is the character type `char`, and one is the `boolean` type for truth values.

The integer types are for numbers without fractional parts. Negative values are allowed. Four integer types are provided: `byte`, `short`, `int`, and `long`. The range of an `int` is just over 2 billion. Bigger integers can be stored in a variable of type `long`. A `byte` requires 1 byte of storage, a `short` 2 bytes, an `int` 4 bytes, and a `long`, 8 bytes.

The floating-point types are used for numbers with fractional parts. There are two floating-point types: `float` and `double`. The range of a `float` is 7 significant digits, and that of a `double` is 15 significant digits. A `float` requires 4 bytes of storage while a `double` requires 8 bytes.

The `char` type uses 2 bytes to store characters represented in the Unicode encoding scheme. Since 16 bits are used to store a character, 2^{16} or 65,536 different characters can be stored in a variable of type `char`. Note that the first 128 characters in Unicode correspond to ASCII characters.

The `boolean` type has two values, `true` and `false`. It can be used for flags which keep track of `true`/`false` conditions. It can also be used for storing the results of logical expressions.

Table 2.1 lists the primitive types in Java.

Name of Type	*Data Stored*	*Size*	*Range of Values*
byte	An integer	1 byte	-128 to 127
short	An integer	2 bytes	-32,768 to 32,767
int	An integer	4 bytes	-2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31} - 1$)
long	An integer	8 bytes	-2^{63} to $2^{63} - 1$
float	A floating point number	4 bytes	Single-precision 32-bit floating point
double	A floating point number	8 bytes	Double-precision 64-bit floating point
char	A single Unicode character	2 bytes	0 to 65,535
boolean	A Boolean value	Is not precisely defined by the Java specification	`true`, `false`

Table 2.1: Primitive Types in Java

2.5 Variables

To declare a variable, the type of the variable must be written first followed by the name of the variable. A variable name must begin with a letter and can be a sequence of letters or digits. Declarations can be placed anywhere in a block of Java code. Here are some examples:

```
int anIntegerVariable;        // declares an integer variable
long aLongVariable;           // declares a long variable
char ch;                      // declares a character variable
float aFraction;              // declares a float variable
int x, y, z;                  // declares several variables at once
```

2.6 Assignments and Initialisations

Assigning a value to a variable is done in the normal way in Java. For example,

```
anIntegerVariable = 56;       // assigns 56 to anIntegerVariable
ch = 'Y';                     // assigns 'Y' to ch
```

In Java, it is possible to declare and initialise a variable on the same line. For example,

```
int i = 10;
char ch = '@';
```

2.7 Operators and Expressions

This section describes the arithmetic operators in Java and shows how to form arithmetic expressions with them. It also describes the comparison and logical operators in Java and shows how to form Boolean expressions with them.

2.7.1 Arithmetic Operators

The usual arithmetic operators + - * / are used in Java for addition, subtraction, multiplication, and division. The / operator denotes integer division if both operands are integers.

Arithmetic expressions are formed by combining literals, variables, and other arithmetic expressions with arithmetic operators. The following are some examples of arithmetic expressions:

```
x + 1
(x - y) * 3
(9.0 * celcius) / 5.0 + 32.0
```

Arithmetic expressions can be used to assign values to variables. For example,

```
x = x + 1;
z = (x - y) * 3;
fahrenheit = (9.0 * celcius) / 5.0 + 32.0;
```

Java provides a shortcut for using arithmetic operators in an assignment statement. For example,

```
x += 4;                      // equivalent to x = x + 4
y -= 1;                      // equivalent to y = y - 1
```

When using the shortcut, the arithmetic operator should be placed to the left of the = sign.

Java provides an operator for integer remainder (also known as the *modulus* operator); it is denoted by %. To find the remainder when 14 is divided by 5, the modulus operator can be used:

```
x = 14 % 5;                  // x is 4 after modulus operation
```

Java has no operator for exponentiation—the built-in `pow()` function in the `Math` library should be used for this purpose. For example,

```
y = Math.pow (x, 5.0);        // y is x raised to the power of 5.0
```

Java has both increment and decrement operators: `x++` adds one to the current value of the variable `x`, and `x--` subtracts one from it. For example,

```
int n = 7;                    // n is assigned the value 7
n++;                          // n now has the value 8
```

2.7.2 Comparison Operators

Comparison operators are used to compare two values. A comparison operator can be used to form a Boolean expression. A *Boolean expression* is an expression which evaluates to either `true` or `false`. Table 2.2 lists the comparison operators in Java and gives some Boolean expressions formed with them. The Boolean expressions in Table 2.2 only contain literals such as 3, 4 and 5. However, a Boolean expression can contain literals as well as variables and arithmetic expressions. For example,

```
(b * b >= 4 * a * c)      // to determine if equation has real roots
```

Comparison Operator	*Java Symbol*	*Boolean Expression*	*Evaluates To*
Less than	<	(3 < 4)	True
Greater than	>	(5 > 5)	False
Equal to	==	(3 == 4)	False
Not equal to	!=	(3 != 5)	True
Less than or equal to	<=	(5 <= 5)	True
Greater than or equal to	>=	(3 >= 4)	False

Table 2.2: Comparison Operators

2.7.3 Logical Operators

Logical operators operate on Boolean values and Boolean expressions. Like comparison operators, logical operators can be used to form Boolean expressions. Table 2.3 lists the three logical operators in Java and gives some Boolean expressions formed with them.

Logical Operator	*Java Symbol*	*Boolean Expression*	*Evaluates To*
And	&&	(3 < 4 && 5 <= 5)	True
Or	||	(5 < 5 || 4 == 5)	False
Not	!	(!(3 == 4))	True

Table 2.3: Logical Operators

An expression with the *and* operator evaluates to `true` if both operands are `true`. An expression with the *or* operator evaluates to `true` if at least one of its operands is `true`. An expression with the *not* operator evaluates to `true` if its operand is `false`; it evaluates to `false` if its operand is `true`.

2.8 Strings

Strings are sequences of characters such as "Hello World". Java does not have a built-in string type. However, the standard Java library contains a predefined *class* called `String`. A variable of type `String` can be declared as follows:

```
String greeting;
```

A variable of type `String` can be used like a primitive type. For example, it can be assigned a value as follows:

```
greeting = "Hello";   // greeting is assigned the string "Hello"
```

Two strings can be concatenated using the + operator. For example,

```
String firstName, lastName, name;

firstName = "John";   // firstName is assigned the string "John"
lastName = "Doe";     // lastName is assigned the string "Doe"

name = firstName + " " + lastName;
                      // firstName is concatenated with lastName
```

After the concatenation operation, the `name` string has the value "John Doe".

The `String` class has several useful methods which can be called to find out information about a string and to manipulate it in various ways. Table 2.4 lists some methods of the `String` class.

Method	*Description*
length()	Returns the number of characters in the given string.
isEmpty()	Returns `true` if the length of the given string is zero.
substring(startIndex, endIndex)	Returns a string that is a substring of the given string (from `startIndex` to `endIndex-1`).
toCharArray()	Returns an array of characters corresponding to the given string.
equals(anotherString)	Returns `true` if `anotherString` is exactly the same as the given string and `false` otherwise.
equalsIgnoreCase(anotherString)	Returns `true` if `anotherString` is the same as the given string, ignoring case considerations; returns `false` otherwise.
toLowerCase()	Returns a new string which consists of the characters in the given string in lower case.
toUpperCase()	Returns a new string which consists of the characters in the given string in upper case.

Table 2.4: Some Useful Methods of the `String` Class

To use the methods in Table 2.4, the name of the variable containing the string must be written first, followed by a dot ("."), followed by the method name and the argument list. For example, to find out the length of the `greeting` string, the `length()` method can be used as follows:

```
int length = greeting.length();
                  // length is 5 since greeting is "Hello"
```

To extract a substring from a larger string, the `substring()` method can be used as follows:

```
String telephone = "1-868-123-4567";
String countryCode = telephone.substring(2, 5);
                  // countryCode is "868"
```

The first argument of the `substring()` method specifies the starting index from which characters are extracted. The second argument specifies the index *one beyond* the last character to be extracted. Note that 0 is the index of the first character in the string.

To test two strings for equality, the `equals()` method should be used instead of the == operator. For example, the following Boolean expression can be used to check if the `countryCode` is 876:

```
(countryCode.equals("876"))
```

To test if two strings are identical except for the upper case/lower case distinction, the `equalsIgnoreCase()` method should be used. For example, if `greeting` is "Hello", "HELLO", "hello", etc., the following Boolean expression evaluates to `true`:

```
(greeting.equalsIgnoreCase("Hello"))
```

2.9 Input and Output

2.9.1 Output using `System.out`

For console-based programs, the `System.out.println()` method can be used to display output on the console. The output is sent to the terminal or to one of the windows in an IDE. Any type of variable can be printed using the `System.out.println()` method. For example,

```
int x = 43;
boolean flag = true;

System.out.println (x);
System.out.println (flag);
```

The output of the above statements is:

```
    43
    true
```

The variable to be printed can be concatenated with a string to produce more descriptive output as follows:

```
System.out.println ("x = " + x);
System.out.println ("Flag is: " + flag);
```

The output of the above statements now becomes:

```
x = 43
Flag is: true
```

Note that `System.out.println()` positions the output cursor to the beginning of a new line after its output has been printed. To print output without advancing the output cursor to a new line, the `System.out.print()` method can be used.

2.9.2 Formatted Output with `System.out`

In the C programming language, the `printf()` function is used with various format strings to control the appearance of output on a console. A similar approach can be taken in Java using the `System.out.printf()` method. For example,

```
int number = 10;
double balance = 2000.00;
String owner = "John Doe";

System.out.printf ("Account Number: %5d ", number);
              // 5 spaces allocated for number

System.out.printf ("Balance: %8.2f ", balance);
              // 8 spaces allocated for balance (2 d.p.)

System.out.printf ("Account Holder: %s\n", owner);
              // prints a string followed by a new line
```

The first argument of the `System.out.printf()` statement is referred to as a *format string*. A format string can contain plain text and/or format specifiers such "%8.2f" (meaning that the variable will be printed with a field width of 8 characters in which two decimal places are reserved for the fractional part). If the actual value of a variable requires fewer characters than the field width, it is padded to left with spaces. The code above generates the following output:

```
Account Number:    10 Balance:  2000.00 Account Holder: John Doe
```

Format strings are beyond the scope of this chapter. You can consult the Java documentation or a C reference for more information.

2.9.3 Input using the `Scanner` Class

One way to obtain user input from the keyboard is to use the built-in `Scanner` class. To indicate that this class will be used, the first line of the source file must contain the following statement:

```
import java.util.Scanner;
            // required in order to use Scanner class
```

To use the `Scanner` class for input, a `Scanner` object must first be created as follows:

```
Scanner scanner = new Scanner(System.in);
```

Next, methods from the `Scanner` class must be used to read specific types of data from the keyboard such as integer values and double values. For example, `nextInt()`, `nextDouble()`, and `next()` must be used to read an integer, a double, and a string from the keyboard, respectively.

The following program uses the `Scanner` class to obtain three values entered by the user at the keyboard (an integer, a double, and a string):

```
import java.util.Scanner;
            // required in order to use Scanner class

public class Input
{
   public static void main(String[] args) {
      int number;
      double balance;
      String owner;

      Scanner scanner = new Scanner(System.in);
            // create Scanner object to input data

      System.out.print ("Please enter the account number: ");
      number = scanner.nextInt();
            // obtain an integer from keyboard input

      System.out.print ("Please enter the account balance: ");
      balance = scanner.nextDouble();
            // obtain a double from keyboard input

      System.out.print
         ("Please enter the name of the account holder: ");
      owner = scanner.next();
            // obtain a string from keyboard input

      System.out.println ("Account Number: " + number);
      System.out.println ("Balance: " + balance);
      System.out.println ("Account Holder: " + owner);
   }
}
```

It is also possible to use the `Scanner` class to read data from a file. However, this is beyond the scope of this chapter.

2.10 Control Structures

2.10.1 Conditional Statements

The general form of the conditional statement in Java is:

```
if (condition)
  { block1 }
else
  { block2 };
```

where *block1* and *block2* contain one or more statements, enclosed in braces. If there is only one statement in a block, it is easier to write that statement without braces, followed by a semicolon. The `else` part is optional; if it is used, it may itself contain nested `if` statements. Note that the condition is written as a Boolean expression.

An `if-then-else` statement to find the maximum of two values `x` and `y` is given below:

```
if (x > y)
   max = x;
else
   max = y;
```

For multiple selections with many alternatives, a `switch` statement is available. The type of the selection variable for the `switch` statement is normally `byte`, `short`, `int`, or `char`. However, a recent release of Java allows the selection variable to be of type `String`.

An example of a `switch` statement is given below:

```
Scanner scanner = new Scanner(System.in);

System.out.print ("Please enter the discount code: ");
int discountCode = scanner.nextInt();

double discount;

switch (discountCode)
{
   case 636:
      discount = 15.0;
      break;

   case 662:
      discount = 20.0;
      break;

   case 672: case 673:
      discount = 25.0;
      break;

   default:
      discount = 10.0;
      break;
}
```

The body of a `switch` statement is called the `switch` block. A statement in the `switch` block can be labeled with a `case` label. The `switch` statement evaluates its expression (the value of `discountCode` in the example above) and then executes the statements that follow the matching `case` label. For example, if the `discountCode` is 636, the `discount` is set to 15.0 percent.

Note that each `case` section must conclude with a `break` statement; otherwise execution of the statements in a matching case label will flow (or fall through) into the next `case` section even if the `case` label of this latter section does not match.

The `default` section is the last section of the case statement and handles all values that are not explicitly handled by one of the `case` sections. For example, a `discountCode` of 681 would end up in the `default` section, resulting in the `discount` being set to 10.0 percent.

Finally, it should be mentioned that a statement in the `switch` block can be labeled with more than one case label (e.g., 672 and 673 in the example above). Thus, the `discount` is set to 25.0 percent if the `discountCode` is 672 or 673.

2.10.2 Repetition

For determinate looping, Java provides a `for` loop. An example of its use is:

```
for (int i=1; i<=10; i++)
   System.out.println (i);    // can be a block enclosed with braces
```

The first slot of the `for` statement (`i=1`) initializes the loop control variable, `i`. The second slot (`i<=10`) specifies the condition that must be true for the loop to continue executing. The third slot (`i++`) is an expression for changing the loop control variable on each iteration of the loop.

For indeterminate looping, the `while` loop statement is provided. The general form is:

```
while (condition)
  { block };
```

The `while` loop version of the `for` loop above can be written as follows:

```
int i = 1;
while (i <= 10) {
   System.out.println (i);
   i++;
}
```

To ensure that the block is executed at least once, the `do` version of the `while` loop can be used:

```
do
  { block }
while (condition);
```

For example, the `do while` version of the `while` loop above is:

```
int i = 1;
do {
   System.out.println (i);
   i++;
} while (i <= 10);
```

2.11 Arrays

Arrays in Java are different from arrays in other programming languages. Three steps are required to create and use an array: declaration, creation, and initialization. To *declare* an array of type `int`, the following syntax should be used:

```
int[] arrayOfInt;           // declares array variable of type int
```

The square brackets after the type `int` declares that `arrayOfInt` is an *array variable* of type `int`. Arrays of the other primitive types can be declared in a similar fashion. To actually *create* the array, the `new` keyword should be used. The follow statement creates the array to hold 100 integer quantities and assigns it to the array variable:

```
arrayOfInt = new int[100];  // creates an array of 100 integers
```

Once the array is assigned to the array variable, the array variable can be treated exactly like an array declared in another programming language. If the array consists of *n* elements, the first element is in location 0 and the last element is in location (*n-1*), similar to an array in C. In the example above, the first element of the array is `arrayOfInt[0]`, and the last element is `arrayOfInt[99]`.

An interesting feature in Java is the ability to create a new array while the program is executing and assign it to an array variable which already "contains" an array. This makes it seem as if the array has been re-sized. For example,

```
arrayOfInt = new int[1000]; // creates a new array of 1000 integers
```

When the above statement is executed, the original array is lost. However, `arrayOfInt` can now store 1000 integers instead of 100.

A `for` loop is commonly used to initialize an array. For example,

```
for (int i=0; i<100; i++)
   arrayOfInt[i] = 0;
```

To find out the size of an array, the `length` *attribute* of the array can be used as follows:

```
int length = arrayOfInt.length;
                            // size is 100
                           // NB: there are no brackets after length
```

It should be noted that the `length` attribute gives the capacity of the array, not the amount of elements that are currently stored. To find out how many elements are currently stored, the program must maintain a count in a separate variable. This variable must be updated accordingly when elements are added to or deleted from the array.

If an array is completely filled, a version of the `for` statement (known as the *foreach* statement) can be used to access all the elements, one at a time. For example,

```
sum = 0;
for (int i: arrayOfInt)
   sum = sum + i;
```

The above code goes through each element `i` in the array and adds its value to `sum`. So, at the end of the loop, `sum` is the total of all the elements in `arrayOfInt`.

Java has a shorthand way to declare, create, and initialize an array at the same time. For example,

```
int[] discountCodes = {636, 662, 672, 673, 681, 682, 725, 727};
```

The above statement causes the `discountCodes` array to be created consisting of 8 elements. The first element of the array is 636 and the last element is 727.

Two-dimensional arrays can also be used in Java. A two-dimensional array can be declared and created as follows:

```
double[][] matrix = new double[5][6];
                       // matrix has 5 rows and 6 columns
```

It is possible to have arrays of objects (i.e., non-primitive types). These types of arrays are discussed in different parts of the book such as in Chapter 3.

Exercises

1. Monica wants to buy a new car. She has saved up a certain amount of money for this purpose but needs to borrow the remaining money from a bank (the principal, *p*) for a certain number of years (the time, *t*) at a certain interest rate (the rate, *r*). The bank will charge Monica interest on the loan using the formula for simple interest:

$$Interest = (p * r * t) / 100$$

The interest calculated is added to the principal and Monica will have to repay the total amount of money in equal monthly installments over the period of the loan.

Write a program that inputs *p*, *r*, and *t* from the user and calculates the interest that will be charged on the loan. Your program should also calculate the monthly installment that Monica will have to pay.

2. Instead of buying a new car, Monica prefers to use public transportation and invest her money in the bank. The bank is offering her an interest rate of *r* if she invests her money for a certain number of years, *t*. Interest will be paid on a monthly basis and compounded.

   Write a program that inputs the principal amount invested (*p*), the interest rate (*r*) and the time in years (*t*) from the user and calculates the total value of Monica's investment after *t* years. Use the following formula for compound interest:

   $$\text{Value of investment after } t \text{ years} = p * [1 + (r / 100) / 12]^{12 * t}$$

   NB: You will need to use the `Math.pow()` function to calculate the exponent, 12 * *t*.

3. Write a program which accepts a student's mark for a course in the range 0 to 100 and calculates the student's grade according to the following scheme:

67- 100:	A
60 - 66:	B+
50 - 59:	B
43 - 49:	C
40 - 42:	D
0 - 39:	Fail

   Your program should generate an error message if the mark is outside the range.

4. Write a program that accepts as input three integer values representing three angles in degrees, and determines if the angles form a triangle. If the angles form a triangle, the type of triangle is determined based on the following categories:

Equilateral:	Three angles are the same
Isosceles:	Two angles are the same
Scalene:	No angles are the same

NB: Three angles form a triangle if their sum is equal to 180 degrees.

5. A leap year is one that is:

   - evenly divisible by 400 or
   - evenly divisible by four and not evenly divisible by 100

   For example, the year 2000 was a leap year since 2000 is evenly divisible by 400. Similarly, the year 2012 was a leap year since 2012 is evenly divisible by four but not by 100. However, the year 2100 will not be a leap year since it does not satisfy any of the two conditions above.

   Write a program that accepts an integer *year* as input and determines if *year* is a leap year.

6. Write a program that accepts as input an integer *year* and an integer *month* in the range 1 to 12 and prints out the name of the month and the number of days in the month. You must cater for leap years.

   Hint: It is easier to use a `switch` statement instead of a series of nested `if` statements.

7. (a) Write a program that accepts as input two integers *m* and *n* and prints a table of squares from *m* to *n* using a `for` loop. For example, if *m* is 2 and *n* is 5, the table should be:

```
Number        Number Squared
======        ==============
2             4
3             9
4             16
5             25
```

   (b) Modify the program in 7(a) to print the table of squares in reverse order from *n* down to *m*, skipping every other number.

   (c) Write the program in 7(a) using a `while` loop and a `do while` loop.

8. A certain tank has *w* litres of water. A fixed *percentage* of water, *p*, is taken out from the tank each day. Write a program that accepts as input *w* and *p*, and starting from the first day, displays the number of the day, the amount of water taken out on that day, and the amount of water remaining in the tank at the end of that day. Your program should stop after 30 days have been displayed or when the amount of water remaining is less than 100 litres, whichever comes first.

   For example, if *w* = 1000, and *p* = 10, the output should start as follows:

```
DAY              AMT TAKEN                AMT REMAINING
===              =========                =============

  1                100.00                   900.00
  2                 90.00                   810.00
  3                 81.00                   729.00
  4                 72.90                   656.10
```

9. Write a program that inputs a set of student marks in an array and then finds the maximum, minimum and average of the set of marks. Your program should also find the amount of marks that were greater than or equal to the average mark.

   The marks must be entered at the keyboard. The number of marks is not known beforehand but input is terminated with a mark of -1. Assume that the marks are in the range 0 to 100.

10. A string is a *palindrome* if it is spelled the same way forwards and backwards. Some examples of string palindromes are: "radar", "able was i ere i saw elba", and "a man a plan a canal panama". Write a program that accepts as input a string *s* and determines if *s* is a palindrome. You should ignore spaces in the string during input.

    Hint: Use a variable *s* of type `String` to store the string entered by the user. Next, convert *s* into an array of characters which can then be examined to find out if the characters form a palindrome. To convert *s* into an array of characters, use the following code:

```
char[] stringChars = s.toCharArray();
            // stringChars is an array of characters
            // corresponding to the input string
```