Chapter 2

The Primaries

INTRODUCTION

In this chapter we will learn the basic constructs of Java, namely the character set, variables, constants, operators, expressions and their usage in drafting simple Java programs.

CHARACTER SET

The set of characters allowed in Java constitutes its character set. All the constituents of a Java program, such as the constants, variables, expressions and statements are written only by using the characters in the following character set:

Numerals : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Alphabets : a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Special characters : + – * / % = < > ( ) . \ | ;, ' : ? # $! ^ & ~ [ ] { }

Java uses the uniform 16-bit coding scheme called **Unicode** to represent characters internally. The *Unicode* can incorporate 65000 characters.

TOKENS

The smallest individual entities in a Java program are known as **tokens**. Five important types of tokens are listed below:

reserved keywords, identifiers, literals, operators, separators

Around 60 words have been reserved as the keywords in Java. They have predefined specific meanings. They should be written in lower-case letters. Here are some valid keywords:

break, continue, switch, private, while

The names that the programmers assign for the classes, methods, variables, objects, packages and interfaces in a Java program are known as **identifiers**. They should follow the following rules:

1. They shall contain digits, alphabets, underscore and dollar sign characters.
2. The first character should not be a digit.
3. They shall have any number of characters.

Some valid identifiers are furnished below:

rectangle, perimeter, count, r2, shape2d, tvChannel

A sequence of valid characters that represents a constant value is called a **literal**. Literals are also known as constants. Five major types of literals available in Java are as follows:

integer, floating point, character, string, boolean

A symbol that represents an operation, such as multiplication or division, is called an **operator**. Some of the well-known operators available in Java are as follows:

```
+    -    *    /    %    <    >    &&   ||   !    ++
--   +=   *=   /=   -=
```

A symbol that is used to separate one group of code from another group is called a **separator**. Some of the well-known separators are listed below:

```
{}   ()   []   ;   ,   '
```

CONSTANTS

An **integer constant** is always a whole number. It can be either positive or negative. This data type is further subdivided into four types, namely **byte, short, int** and **long** on the basis of their sizes. The minimum and maximum possible values for each one of these four data types are furnished in the following table:

Table 2.1 *Integer Data Types*

Type	Size	Minimum Possible Value	Maximum Possible Value
byte	1 byte	–128	127
short	2 bytes	–32, 768	32, 767
int	4 bytes	–2, 147, 483, 648	2, 147, 483, 647
long	8 bytes	–9, 223, 372, 036, 854, 775, 808	9, 223, 372, 036, 854, 775, 807

A **real constant** will always have a fractional part. This data type is further subdivided into two types, namely **float** and **double.** The minimum and maximum possible values for these two data types are furnished in the following table:

Table 2.2 *Real Data Types*

Type	Size	Minimum Possible Value	Maximum Possible Value
float	4 bytes	1.4012e – 45	3.4028e + 38
double	8 bytes	4.9406e – 324	1.7976e + 308

Two possible boolean constants are **true** and **false**. A single character that is written inside single quotation marks is defined as a character constant. The default size that is made available for a **char** type constant is 2 bytes. A ***char*** type variable can store only a single character. Here are some valid character constants:

```
'c',   '6',   '+',   '%'
```

Characters have numeric values. For example, the character 'A' is actually encoded as 65. We can use some special backslash character constants inside character constants. The backslash characters are used as ***escape*** characters. The character sequence **\n** is called an ***escape sequence***. The escape sequences are used to encode certain special category of characters that will otherwise be difficult to include in a string. The important escape sequences are listed in the following table:

Table 2.3 Backslash Character Constants

Backslash character constants	Meaning
\b	backspace
\n	new line
\t	tab
\r	carriage return
\f	form feed
\\	backslash
\'	single quote
\"	double quote

Each backslash character constant contains the backslash character \ followed by a character. Even though each one of the above escape sequences seems to involve two keystrokes, each one of them is treated as a character constant in Java. The backslash character constants are also known as **escape sequences**.

A string of valid characters enclosed by double quotation marks is defined as a **string constant**. Here are some valid string constants:

```
"Welcome to Java"    "2004"    "02-01-1961"    "E-Commerce!"
```

The string constants are useful in representing various types of non-numeric data, such as dates, times, names and addresses.

VARIABLES

A **variable** is an **identifier** that denotes a storage location in which a value is stored. While the constants remain unchanged throughout the execution of a program, a variable may take different values during the execution of a program. To name our variables, here are some rules that we must follow:

1. The first character in a variable name should not be a digit.
2. A variable name may consist of alphabets, digits, the underscore character and the dollar character.
3. A keyword should not be used as a variable name.
4. White spaces are not allowed within a variable name.
5. A variable name can be of any length.

Some valid variable names are listed below:

```
total,    average,    root_2,    branch_code
```

Since Java is case-sensitive, uppercase letters and lowercase letters are distinct in Java. Thus the variable name **sum** is not the same as the variable name ***Sum***.

Every variable has a **data type.** The four basic data types are **integer, real, character** and **boolean**. A variable's data type specifies the size and type of value that can be assigned to that variable.

Every variable is the name of a storage location. Since the **size** depends on the data type of a variable, a storage location that stores an **int** type variable has to be of length 4 bytes, while a storage location that stores a **double** type value has to be of length 8 bytes.

The Java compiler fixes the size of a storage location and the type of value to be stored in it on the basis of the information it receives from a **type declaration statement** of the following form:

```
access-specifier    data-type    variable-name
```

Here,

1) **access specifier** specifies which methods can access the variable. Most variables are declared as private. In such a case, all the methods in the class in which the variable is declared can access this variable. We will learn more about access specifiers in chapter seven.

2) **data-type** refers to the type of the variable. Here is a valid **type declaration statement**:

```
int     mark ;
```

This statement informs the Java compiler that the variable **mark** is of ***int*** type and the storage location that stores the value of ***mark*** should be of length 4 bytes. Some more valid *type declaration statements* are furnished below:

```
float   fahrenheit, centigrade;
double  radius;
char    letter, c;
boolean flag1, flag2,flag3;
long    file_length;
short   count;
```

OPERATORS AND EXPRESSIONS

Operators are used to perform computations. The four basic arithmetic operations, namely the addition, subtraction, multiplication and division, are denoted in Java by the arithmetic operators +, –, * and /. The % operator is used to compute the remainder of a division.

When the constants and variables are combined with the arithmetic operators, arithmetic expressions are formed. Here are some valid arithmetic expressions:

```
salary + allowance,     total - discount,     sales * commission_rate,
                                              sum/count, 11 % 2
```

The **unary minus operator, increment** and **decrement** operators, namely ++ and — are also available in Java.

Java supports bitwise operators, which are used for testing the bits or shifting them to the left or right. These operators always operate on integer operand(s). The bitwise operators and the operations they represent are listed in Table 2.4.

Table 2.4 *Bitwise Operators*

Bitwise Operators	Represented Operation
!	bitwise OR
&	bitwise AND
^	bitwise exclusive OR
~	one's complement
>>	right shift
<<	left shift
>>>	right shift with zero fill

The arithmetic operations involved in an arithmetic expression are carried out in the following order of precedence:

First Level	:	Unary minus + + – –
Second Level	:	* / %
Third Level	:	+ –

The operators with the first level of precedence (i.e. unary minus, ++, – –) are applied before those with the second level of precedence (i.e. *, /, %) are applied. Similarly, the operators with the second level of precedence are applied before the operators with the third level of precedence (i.e. +, –) are applied.

All the operators, their description, associativity and precedence level are furnished in Table 2.5.

Table 2.5 Operators and Precedence Levels

Operator	Represented Operation	Associativity	Precedence Level
[ ]	Array element reference	Left to right	1
( )	Function invocation		
	Object member selection		
++	Increment	Right to left	2
–	Unary minus		
–	Decrement		
!	Logical negation		
~	One's complement		
(data type)	Casting		
%	Modulus	Left to right	3
*	Multiplication		
/	Division		
+	Addition	Left to right	4
–	Subtraction		
>>	Right shift	Left to right	5
<<	Left shift		
>>>	Right shift with zero fill		
>	Greater than	Left to right	6
>=	Greater than or equal to		
<	Less than		
<=	Less than or equal to		
instance of	Type comparison		
==	Equality	Left to right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional operator	Right to left	13
=	Assignment operator	Right to left	14
operator =	Shorthand assignment	Right to left	14

When we assign an expression to a variable, the *types* of the variable and the expression should be compatible. For example, the following statements will generate an error as a floating-point expression cannot be assigned to an integer variable.

```
double  sum = 10.1;
int total   = sum;
```

We must convert the floating-point expression to integer by using a *cast*, as shown below:

```
int    total  =  ( int ) sum;
```

Here, *cast* (int) converts the floating-point value of the variable *sum* to an integer and then assigns it to the integer variable *total*. Here, the value of the variable *sum* is 10.1 and the value of the variable *total* is 10. Thus, there is a loss of information when *cast* is used. We convey to the system that we agree to this information loss, when we use *cast*. Whenever there is the possibility of information loss, we have to use *cast* in Java. A *cast* always has the form (datatypeName). Two examples are (int) and (float).

Sometimes, we may wish to *round up* a floating-point value when its fractional part is 0.5 or larger. We shall use the *round( )* method in the predefined *Math* class for this purpose, as shown below:

```
double sum    =  23.6;
int    total  =  ( int ) Math.round ( sum );
```

Here, the *round( )* method rounds up the value 23.6 to 24.0. The *cast*(int) converts it into 24 and then assigns it to the integer variable *total*.

LIBRARY METHODS

Library methods are normally used to write the expressions in a compact manner. For example, the expression

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

can be compactly represented as shown below, by using the **sqrt( )** library method:

```
(- b + Math.sqrt ( b * b - 4 * a * c)) / ( 2 * a)
```

All the important mathematical methods are listed in the **Math** class. Table 2.6 briefly describes these methods:

Table 2.6 Mathematical Methods

Library Method	Purpose
Math.sqrt(x)	To find square root of x (≥0)
Math.pow(x,y)	To find x^y (x > 0 or x = 0 and y > 0 or x < 0 and y is an integer)
Math.sin(x)	To find Sine of x (x in radians)
Math.cos(x)	To find Cosine of x
Math.tan(x)	To find Tangent of x
Math.asin(x)	To find Arc sine of x
Math.acos(x)	To find Arc cosine of x
Math.atan(x)	To find Arc tangent of x
Math.toRadians(x)	To convert x radians to degrees
Math.toDegrees(x)	To convert x degrees to radians

Library Method	Purpose
Math.exp(x)	To find e^x
Math.log(x)	To find natural log
Math.round(x)	To find closest integer to x
Math.ceil(x)	To find smallest integer ≥ x
Math.floor(x)	To find largest integer ≤ x
Math.abs(x)	To find the absolute value |x|

STRINGS

Next to the numeric data type, *string* is the most important data type used in programs. A *string* is a sequence of characters. In Java, a string is enclosed in quotation marks, as in the case of the example "Vijay Nicole Imprints Pvt Ltd". But the quotation marks are not themselves part of the string.

The numeric data types are specified by the keywords *int, double*, etc. While these keywords start with a lowercase letter, the keyword *String*, which represents the *string* data type, starts with an uppercase letter. This is because of the fact that the keyword *String* is the name of a predefined class.

The number of characters in a string is provided by the *length( )* method of the *String* class, as illustrated below:

```
String publisher = "Vijay Nicole Imprints Pvt Ltd" ;
int wordLength  = publisher.length( );
```

Since the string "Vijay Nicole Imprints Pvt Ltd" consists of 29 characters, the *length( )* method will return the value 29.

A string that contains no characters is called an *empty string*. Its length is zero and it is written as " ".

Two strings shall be concatenated by using the + operator. This operator is a powerful operator. If *one of the two operands*, either to the left or the right of the + operator is a string, then the other operand is automatically converted into a string and both strings are concatenated. Here is an example:

```
String  s      =  "Price :" ;
int     price  =  83;
String  result =  s + price;
```

Here, the expression *s + price* has a string operand and an *integer* operand. They will be concatenated into the single string *"Price : 83"* and assigned to the variable *result*.

The *substring( )* method in the *String* class helps us to extract a substring from a given string. Here is a simple example:

```
String registerNumber  =  "2003CSE29";
String branch          =  registerNumber.substring(4,3);
```

The index of the first character in a string is always *zero*. In the above example, the *substring( )* method has the two arguments *4* and *3*. While the first argument specifies the starting position of the substring to be extracted, the second argument specifies the number of characters to be retrieved from the starting position. Thus in the above example, the *substring( )* method will return the substring "CSE". We will learn more about methods in the *String* class in a later chapter.

I/O (INPUT/OUTPUT) STATEMENTS

Every program needs some data as input. Similarly, every program provides some processed data as output. While the input operations enable us to provide input data to a program, the output operations help us to get the output displayed in a desired format. Java provides a rich set of I/O functions for performing a variety of operations.

Console Output

The **out** object in the **System** class represents the standard output stream. The *out* object has two methods for sending the output to the *Console* screen. They are the **print( )** and the **println( )** methods. While the *print( )* method displays the output without a newline character, the *println( )* method displays the output with a newline character. Since both the *print( )* and the *println( )* methods are heavily overloaded methods, they can output many different types of data, such as int, double, char, bool and String.

Consider the following program segment:

```
System.out.print ("Welcome to Java.");
System.out.println ("Have a nice time." );
```

Here, the System.out.print( ) method displays the message **Welcome to Java** on the screen and retains the cursor on the same line. The System.out.println( ) method displays the string **Have a nice time** and moves the cursor to the beginning of the next line. Here is a sample output:

```
Welcome to Java.Have a nice time.
```

Sometimes, we may wish to have a blank line between two sets of output values. We shall print a blank line by using a *println( )* method, as shown below:

```
System.out.println ("Value of A = 1000");
System.out.println ( );
System.out.println ("Value of B = 2000");
```

The above snippet will render the output in the following form:

```
Value of A = 1000

Value of B = 2000
```

Keyboard Input

In Java, any input is read in as a **string**. In the following snippet, the name that is keyed in by the user is received by the system and stored in the buffer **System.in** in terms of bytes. When the second statement in the following snippet is executed, the bytes in the buffer *System.in* are converted into characters and stored in the object *reader*. An *InputStreamReader* object, such as *reader*, can read characters, but it cannot read a whole string at a time. In order to overcome this limitation, we turn the input stream *reader* into a *BufferedReader* object, by using the *third* statement in the following snippet. When the *fourth* statement in the following snippet is executed, the *readLine( )* method reads in a string (i.e. a name) and stores it in the variable *name*. We will learn a lot about I/O streams, which are involved in the above discussion, in chapter 14.

```
System.out.print ( "Enter Your Name :" ) ;
InputStreamReader reader = new InputStreamReader (System.in );
BufferedReader in = new BufferedReader ( reader );
String name = in.readLine( );
```

Suppose that we provide 1000 as the input to the following snippet. This value will be initially read in as the string "1000" and stored as the value of a string type variable **text**. The **parseInt( )** method, which is available

in the ***Integer*** class, takes out the integer 1000 from the string "1000" and stores it as the value of the *integer* type variable **dollar**. This is the standard way of providing an *integer value* as input to a Java program.

```
System.out.print ( "Enter the amount in dollars:" ) ;
InputStreamReader reader = new InputStreamReader ( System.in );
BufferedReader in = new BufferedReader ( reader );
String text = in.readLine ( );
int dollar = Integer.parseInt( text ) ;
```

The first four statements in the following snippet are same as the first four statements in the above snippet. Any input is read in as a ***string*** in Java. Suppose that we provide 6.5 as the input to the following snippet. This value will be read in as the string "6.5" and stored as the value of the string type variable **text**. The **parseDouble( )** method in the ***Double*** class takes out the fractional value 6.5 from the string "6.5" and stores it as the value of the *double* type variable **r**. This is the standard way of providing a *fractional value* as input to a Java program.

```
System.out.print ( "Enter the radius of a sphere :" ) ;
InputStreamReader reader = new InputStreamReader ( System.in ) ;
BufferedReader in = new BufferedReader ( reader ) ;
String text = in.readLine ( ) ;
double r = Double.parseDouble ( text ) ;
```

SIMPLE PROGRAMS

As we now know the concepts of variables, constants, expressions and simple I/O statements, we shall now develop some simple Java programs that make use of these concepts.

Example 2.1 The following program reads in the name of a person and then displays the same.

```
import java.io.*;
class HelloName
 {
 public static void main (String[ ] args) throws IOException
  {
    System.out.print ( "Enter Your Name  :" ) ;
    InputStreamReader reader = new InputStreamReader( System.in );
    BufferedReader in = new BufferedReader ( reader );
    String name = in.readLine( );
    System.out.println ( "Your Name is     : " + name ) ;
  }
}
```

Output

```
Enter Your Name  :  Vignesh
Your Name is     :  Vignesh
```

As we may recall from the previous chapter, each simple Java program is written in the form of a class. The above program is written as the class with the name **HelloName**. This class consists of only the **main( )** method. The main purpose of the above program is to read in the name of a person and then display it on the Console Screen.

The name that is keyed in by the user is received by the system and stored in the buffer **System.in** in terms of bytes. When the second statement in the main( ) method of the above program is executed, the bytes in the buffer *System.in* are converted into characters and stored in the object *reader*. An *InputStreamReader* object,

such as *reader*, can read characters, but it cannot read a whole string at a time. In order to overcome this limitation, we turn the input stream *reader* into a *BufferedReader* object, by using the third statement in the above *main( )* method:

```
BufferedReader in = new BufferedReader(reader);
```

Then, we use the readLine( ) method in the fourth statement, to read in a string (i.e. a name).

```
String name = in.readLine( );
```

At this stage, the String type variable *name* will have the name that is keyed in by the user. Finally, the content of the String type variable *name* will be displayed on the console screen, when the last statement in the main( ) method is executed.

Here is the pictorial representation of the activities that take place in the above program.

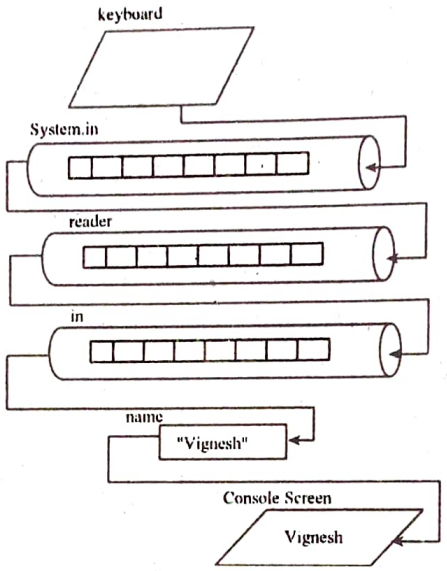

***Fig 2.1** Steps Involved in Providing an Input Value*

Java allows the use of single line comments and multi-line comments. A single line comment starts with // and ends at the same line as shown below:

```
// The following program will read in a name
```

A multi-line comment spreads over several lines. Such a comment begins with /* and ends with */. Here is an example:

```
/* This program will receive a name from the user
   and then display it on the console screen
*/
```

An **error** is known as an *exception* in Java. If a problem is encountered while reading input, the *readLine( )* method will generate an exception of type *IOException*. Java Runtime insists that we either handle this exception or inform the Runtime that we are not handling this exception. We will learn a lot in Chapter 12 about handling exceptions. For the time being, we will inform the Runtime that we are not handling the exception. As a result, the Runtime itself will handle the exception, if it occurs.

In the above program, the *readLine( )* method, which may generate an exception of type *IOException*, is invoked within the *main( )* method. In order to inform the Runtime that we are not handling the exception, we have to tag the *main( )* method with a *throws* specifier, as shown below:

```
public static void main (String[ ] args) throws IOException
```

Example 2.2 The following program reads in an amount in dollars and then converts it into the corresponding amount in rupees.

```
// Program for currency conversion
import java.io.*;
class CurrencyConversion
{
    public static void main ( String[ ] args ) throws IOException
    {
        System.out.print ( "Enter the amount in dollars : " ) ;
        InputStreamReader reader = new InputStreamReader( System.in );
        BufferedReader in = new BufferedReader ( reader );
        String text = in.readLine ( );
        int dollars = Integer.parseInt( text ) ;
        int rupees = 47 * dollars ;
        System.out.println( "Amount in dollars    : "  +  dollars ) ;
        System.out.println ( "Amount in rupees    : "  +  rupees ) ;
    }
}
```

Output

```
Enter the amount in dollars :   1000
Amount in dollars           :   1000
Amount in rupees            :   47000
```

The first four statements in the *main( )* method of the above program are same as the first four statements in the *main( )* method of the previous program. In Java, any input is read in as a *string*. Suppose that we provide 1000 as the input to the above program. This value will be read in as the string "1000" and stored as the value of a String type variable **text**. The **parseInt( )** method takes out the integer 1000 from the string "1000" and stores it as the value of the integer type variable ***dollar***.

While the sixth statement in the *main( )* method converts amount in dollars into rupees, the last two statements display the given amount in dollars and its equivalent amount in rupees.

Example 2.3 The following program reads in the radius of a sphere and then computes its volume.

```
// Program to compute the volume of a sphere
import java.io.*;
class SphereVolume
{
    public static void main (String[ ] args) throws IOException
    {
        System.out.print ( "Enter the radius of a sphere :" ) ;
        InputStreamReader reader  =  new
        InputStreamReader(System.in) ;
        BufferedReader  in  =  new  BufferedReader ( reader ) ;
        String text  =  in.readLine ( ) ;
        double  r  =  Double.parseDouble ( text ) ;
        double volume = (4.0 / 3.0) * 3.14 * r * r * r;
        System.out.println ( "Radius of the sphere     :  " +  r ) ;
        System.out.println ( "Volume of the sphere     :  " +  volume ) ;
    }
}
```

Output

```
Enter the radius of a sphere    : 45
Radius of the sphere            : 45.0
Volume of the sphere            : 381510.0
```

The first four statements in the *main( )* method of the above program are same as the first four statements in the *main( )* method of the previous program. Any input is read in as a *string* in Java. Suppose that we provide 6.5 as the input to the above program. This value will be read in as the string "6.5" and stored as the value of the String type variable ***text***. The ***parseDouble( )*** method takes out the fractional value 6.5 from the string "6.5" and stores it as the value of the double type variable **r**.

While the sixth statement in the *main( )* method computes the volume of the sphere, the last two statements display the given radius and its corresponding volume.

FORMATTING THE OUTPUT VALUES

Sometimes, we may wish to format the output values before displaying them on the console screen. For example, we may want an output value to be displayed with two digits after the decimal point, as shown below:

```
Price = 98.75
```

We shall achieve this by using the methods in the NumberFormat class, as illustrated below:

```
NumberFormat f = NumberFormat.getNumberInstance( );
f.setMaximumFractionDigits(2);
System.out.println( "Price = " + f.format( price ) );
```

The setMaximumFractionDigits(2) method sets the maximum number of fraction digits to 2. So, the values are rounded to two digits. For example, 89.246 will be rounded to 89.25. But, 89.198 will be rounded to 89.2, not 89.20. If we wish to have trailing zeroes, we have to also set the minimum number of fraction digits to 2, by using the setMinimumFractionDigits( ) method, as shown below:

```
f.setMinimumFractionDigits( 2 );
```

Example 2.4 The following program formats a fractional value in such a way that the fractional part will have exactly two digits. It ensures that the trailing zero is also retained.

```
import java.text.*;
class NumberFormatDemo
{
    public static void main(String[ ] args)
    {
        NumberFormat f = NumberFormat.getNumberInstance( );
        f.setMaximumFractionDigits(2);
        f.setMinimumFractionDigits(2);
        System.out.println( "Price = Rs." + f.format(89.198) );
    }
}
```

Output

```
F:\Java\JavaPro\ch2>java NumberFormatDemo
Price = Rs.89.20
```

Chapter 3

Control Statements

INTRODUCTION

The simplest programs just contain one massive list of instructions that the computer follows one after another. Only such types of programs have been discussed in the previous chapter. However, in many programming situations, there arises a need for us to exercise control over the order of execution of the statements present in a Java program. A special type of statement available in Java, namely **control statements** enables us to achieve this objective.

The two major types of *control statements* available in Java are the **decision-making statements** and the **repetitive statements.** The **if** statement, and the **switch** statement are the two important types of *decision making statements.* The **while** statement, the **for** statement and the **do...while** statement are the three important types of *repetitive statements.* All these statements are discussed in this chapter with suitable examples.

'if' STATEMENT

This statement enables us to take decisions based on a condition. Such a condition is expressed in the form of a relational expression or a logical expression. The value of a relational or logical expression is always one of the two logical constants, namely, **true** and **false**. If the value of the relational or logical expression involved in an **if** statement is true, one action is taken; otherwise, another action is taken.

Here is the correct syntax of the **if** statement:

```
if ( test-expression )
        statement-block-1
else
        statement-block-2
```

Here, 1. **if** and **else** are keywords.

2. **test-expression** is a relational or logical expression.
3. **statement-block-1** may be a single statement or a group of statements.
4. **statement-block-2** may be a single statement or a group of statements.

The function of the **if** statement is simple. If the value of **test-expression** is **true**, **statement-block-1** will be executed. Otherwise, **statement-block-2** will be executed.

Here are some valid **if** statements:

```
1)  if  ( amount  >  20000 )
        tax  =  0.10  *  amount ;
```

```
    else
        tax = 0.05 * amount ;
2)  if ( gross_income > 200000 )
    {
       income = 0.25 * gross_income ;
       profession_tax = 0.07 * gross_income ;
    }
    else
    {
       income_tax = 0.15 * gross_income ;
       profession_tax = 0.05 * gross_income;
    }
3)  if ( mark >= 75 )
       count++ ;
```

Java allows many abbreviations. For example, consider the following **if** statement:

```
if ( a > b )
   c = a ;
else
   c = b ;
```

The function of the above ***if*** statement is the same as the function of the following *assignment statement* that involves the selection *operator*.

```
c = ( a > b ) ? a : b ;
```

Here, if the condition **a > b** is **true**, then the value of **a** will be assigned to **c**; otherwise, the value of **b** will be assigned to **c**. The operator ? is also known as **ternary operator** or **conditional operator**.

Example 3.1 In the following program, the age of a person is read in. His age is then compared with 18 through an ***if*** statement. He is declared to be eligible to cast his vote only if his age is greater than or equal to 18.

```
//Program to illustrate the syntax of IF statement
import java.io.*;
class Voting
{
 public static void main ( String[ ] args ) throws IOException
 {
   System.out.print( "Enter your age in completed years : " );
   InputStreamReader reader = new InputStreamReader (System.in) ;
   BufferedReader in = new BufferedReader (reader) ;
   String text = in.readLine ( );
   int age = Integer.parseInt (text);
   if ( age >= 18 )
     System.out.println("You are eligible to cast your vote " );
   else
     System.out.println("You have to wait for " + ( 18-age ) + " years " );
 }
}
```

Output

```
Enter your age in completed years : 28
You are eligible to cast your vote
Enter your age in completed years : 12
You have to wait for 6 years
```

Example 3.2 In the following example, the interest rate is fixed through a *nested if* statement on the basis of the schedule given below. Then, the simple interest is computed.

Principal	Period of deposit	Interest Rate
<= 10000	<= 2 years	9%
<= 10000	> 2 years	10%
>10000	–	11%

```
// Program to illustrate the structure of Nested IF statement
import java.io.*;
class Interest
{
    public static void main ( String[ ] args ) throws IOException
    {
        InputStreamReader reader = new InputStreamReader ( System.in ) ;
        BufferedReader in = new BufferedReader ( reader ) ;
        System.out.print( "Enter the Deposit Amount : " );
        String text = in.readLine( );
        int p = Integer.parseInt ( text );
        System.out.print( " Enter the period of Deposit : " );
        text = in.readLine( );
        int n = Integer.parseInt ( text );
        double rate;
        if ( p > 10000 )
            rate = 0.11 ;
        else
            if ( n > 2 )
                rate = 0.10 ;
            else
                rate = 0.09 ;
        double interest = p * n * rate ;
        System.out.println ( " Interest to be paid = " + interest );
    }
}
```

Output

```
Enter the Deposit Amount    : 10000
Enter the period of Deposit : 3
Interest to be paid         = 3000.0
```

'switch' STATEMENT

The main problem with using a nested **if** statement is that such a statement is hard to read and understand and cumbersome to write. The practical alternative is to use another popular control statement called the **switch** statement in such situations. When a ***switch*** statement is executed, the program's control branches out to one of the many specified statements.

Here is the syntax of this statement.

```
switch ( expression )
{
    case value-1 : statement-block-1 ;
```

```
    case  value-2    :   statement-block-2 ;
    .
    .
    .
    case  value-n    :   statement-block-n ;
    default          :   default-block ;
}
```

Here, 1. **switch, case** and **default** are keywords.

2. **expression** is any valid expression that yields an integer constant or character constant.
3. Each one of the entries value-1, value-2,..., value-n is an integer constant or expression, such as 6 * 3.
4. Each **statement block** may be single statement or a group of statements.
5. **default** part is optional.

The function of the ***switch*** statement is simple. First, the expression following the keyword ***switch*** is evaluated. Its value is then compared with value-1, value-2,..., value-n, one at a time, starting from value-1. If the value of the expression matches with value-1, statement-block-1, statement-block-2,..., and statement-block-n will be executed. If the value of the expression matches with value-2, statement-block-2, statement-block-3,..., statement-block-n will be executed. If the value of the expression matches with value-n, statement-block-n will be executed.

However, the execution of the statements following the subsequent labels, as stated above, can be avoided by appropriately using the **break** statement.

If the value of the **expression** following the keyword ***switch*** does not match with any of the labels, then the statement following the keyword **default** will be executed.

Here is a simple example. The following ***switch*** statement shall be used to find whether the given lower-case alphabet is a vowel or not.

```
char  letter ;
switch  ( letter )
{   case  'a'   :
    case  'e'   :
    case  'i'   :
    case  'o'   :
    case  'u'   :  System.out.println( " The given alphabet is a vowel ");
                   break ;
    default     :  System.out.println( " The given alphabet is not a vowel ");
 }
```

Example 3.3 In the following example, it is assumed that there are three categories of employees in a company. It is also assumed that these three categories of employees are provided allowances at three different rates. A ***switch*** statement is used to determine the **allowance rate** for an employee on the basis of his **category**. The allowance is then computed.

```
//Program to illustrate the use of SWITCH statement
import  java.io.* ;
class  Tax
 {
  public static void main (String[ ] args) throws IOException
```

```
    {
     InputStreamReader  reader  =  new InputStreamReader ( System.in ) ;
     BufferedReader    in    =  new BufferedReader ( reader ) ;
     System.out.print ( "Enter the Income          : " );
     String   text1  =  in.readLine ( ) ;
     int   income   = Integer.parseInt ( text1 ) ;
     System.out.print ( "Enter the Employee Category  : " );
     String   text2  = in.readLine ( );
     int  emp_catagory = Integer.parseInt ( text2 ) ;
     double  allowance_rate  =  0 ;
     switch ( emp_catagory  )
     {
         case  1  :allowance_rate = 0.05 ;
                   break;
         case  2  :allowance_rate = 0.07 ;
                   break;
         case  3  :allowance_rate = 0.10 ;
                   break;
      }
     double   allowance  =  income * allowance_rate ;
     System.out.println( "Allowance to be paid          : " + allowance );
    }
}
```

Output

```
Enter the Income           :  10000
Enter the Employee Category :  1
Allowance to be paid       :  500.0
```

'while' STATEMENT

In many programming situations, there arises a need for certain instructions to be executed over and over again. A special type of statements called the **repetitive statements** shall be used in such situations. The **while** statement, the **for** statement and the **do...while** statement are the ***repetitive statements*** available in Java. While the **while** statement is discussed in this section, the other two statements are explained in the next two sections.

The execution of a ***while*** statement results in the repeated execution of a **simple** or **compound** statement as long as a **specified** condition remains **'true'**.

Here is the correct syntax of this statement:

```
while ( test-condition )
        statement-block ;
```

Here, 1. **while** is a keyword.

2. **test-condition** is a logical expression.

3. **statement-block** may be a single statement or a group of statements.

When a ***while*** statement is executed, the value of the **test-condition** following the keyword **while** will be found. If it is **'true'**, the **statement-block** following the ***test-condition*** will be executed. Then the value of the ***test-condition*** will be found again. If it is **'true'**, the **statement-block** will be executed again.

This simple cycle of evaluation of the *test-condition* and the **execution** of the statement-block will continue until the value of the *test-condition* becomes **'false'**. Once the value of the ***test-condition*** becomes **'false'**, control will come out of the **while** loop and it will pass on to the first statement that follows the ***while*** statement.

The function of the ***while*** statement shall be pictorially represented as follows:

Consider the following program segment:

```
j = 1;
while ( j <= 4 )
{
    System.out.println( j ) ;
    j = j + 1 ;
}
```

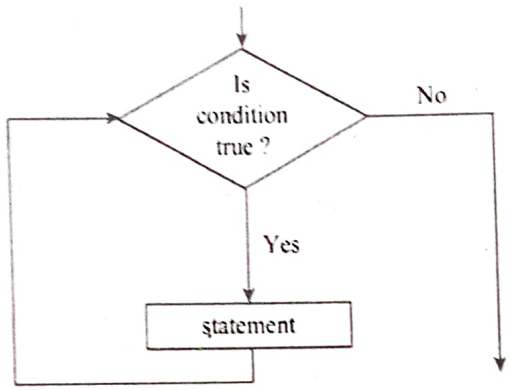

Fig. 3.1 *'While' Statement*

Here, the compound statement that follows the condition **j <= 4** will be repeatedly executed as long as the value of the condition **j <= 4** remains **true**. The value of this ***condition*** will be **true** for j = 1, 2, 3 and 4. Thus the above program segment will generate the values 1, 2, 3 and 4 as the output.

A ***while*** statement results in the repeated execution of the statement associated with it. Due to this reason, the ***while*** statement is said to form a loop. The **statement** that is associated with the ***while*** statement constitutes the **body of the loop**. For example, the body of the *while* statement specified above is as follows:

```
{
    System.out.println( j ) ;
    j = j + 1 ;
}
```

Sometimes, in the middle of execution of a ***while*** statement, we may wish to bring the control out of the ***while*** loop. We shall use a special statement known as the *break* statement in such situations. As soon as a **break** statement is encountered during the execution of a ***while*** loop, the program's control will come out of the entire ***while*** loop and it will pass on to the first statement that follows the ***while*** statement.

The simple form of the *break* statement has already been discussed. In Java, there is a second form of the *break* statement. Its form is

break label;

When a *break* statement of the above form is executed, control will jump to the *end* of the statement that is tagged with the label. Any statement (*if* statement, *while* statement, etc.,) can be tagged with a label. Here is an example.

```
looplabel:
while ( n <= 10 )
{
    .....
    if ( term < 0.001 )
    break looplabel ;
    .....
}
```

If the condition *term < 0.001* in the above example becomes true at any stage during the execution of the *while* loop, the control will jump to the *end* of the *while* loop. In such a case, execution will resume from the first statement that follows the *while* statement.

However, in some situations, we may not like to jump out of a ***while*** loop, but we may wish to jump back to the **condition** at the top of the ***while*** loop. A special statement known as **continue** statement shall be used in such situations.

Example 3.4 In the following program, a positive whole number is first read in. Then the number of digits in that number is found with the help of a ***while*** statement. It shall be noted here that this ***while*** statement is capable of finding the number of digits in a number with any number of digits.

```
//Program to illustrate the use of WHILE statement
import java.io.*;
class Count
{
   public static void main ( String[ ] args ) throws IOException
   {
      InputStreamReader reader = new InputStreamReader ( System.in ) ;
      BufferedReader in  =  new BufferedReader ( reader ) ;
      System.out.print(" Enter a whole Number : ");
      String  text  =  in.readLine ( ) ;
      int  number = Integer.parseInt (text) ;
      int  count = 0;
      while (number != 0)
      {
       number = number / 10 ;
       count = count + 1 ;
      }
   System.out.println("The Given Number is a " + count + " Digit Number");
   }
}
```

Output

```
Enter a whole Number : 123
The Given Number is a 3 Digit Number
```

'do . . while' STATEMENT

The **test-condition** is checked at the top of the loop in the case of the **while** statement. On the other hand, the *test-condition* is checked at the bottom of the loop in the case of the another repetitive statement, known as the

do..while statement. But for this minor difference, the function of the ***do..while*** statement is same as that of the ***while*** statement. The syntax of this statement is as follows:

```
do
   statement-block
while (test-condition) ;
```

Here, 1) **statement-block** may be a single statement or a group of statements.
2) **test-condition** is a logical expression.
3) **do** and **while** are keywords.

The **statement-block** that follows the keyword **do** will be repeatedly executed as long as the value of the **test-condition** remains **true**. Once the value of this **test-condition** becomes **false**, the program's control will pass on to the first statement that follows this loop.

Since the ***condition*** is tested only at the bottom of the ***do..while*** loop, the ***statement*** will be executed atleast once. The function of the *do..while* statement is pictorially illustrated below:

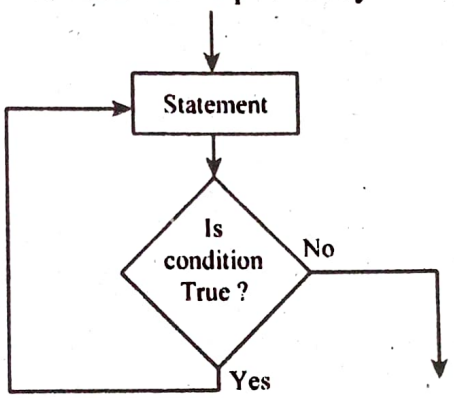

Fig 3.2 *'do . . while' Statement*

As in the case of the ***while*** loop, the **break** statement shall be used to bring the control out of the ***do..while*** loop. When the ***continue*** statement is encountered in a *do..while* loop, control will jump straight to the ***condition*** which is at the bottom of the *do..while* loop.

The following ***do..while*** statement shall be used to extract the digits in a positive whole number. The use of the *do..while* statement is appropriate in this example as a positive whole number will contain atleast one digit.

```
do
  { digit  = number % 10 ;
    System.out.println( digit ) ;
    number  =  number / 10 ;
  }
  while ( number != 0 ) ;
```

'for' STATEMENT

If we wish a **statement** (simple or compound) to be repeatedly executed as long as a condition remains '**true**', then we shall use the ***while*** statement or the **do..while** statement. But if we already know how many times we want a statement to be repeatedly executed, then the appropriate statement to be used is the ***for*** statement. Here is the syntax of this statement:

```
for( statement1; test-condition; statement2)
    statement-block;
```

Here, 1. **statement1** is usually an initialization statement, such as j = 1.
2. **test-condition** is a logical expression, such as j <= 10.
3. **statement2** is usually an increment statement, such as j = j + 1.
4. **statement-block** may be a single statement or a group of statements.

The function of the **for** statement is simple. The ***statement1*** is executed only once in the beginning. Then the value of *test-condition* is found. If it is ***true***, then the ***statement-block*** is executed. Then, ***statement2*** is executed and the value of **test-condition** is checked again. If it is true, then ***statement-block*** is executed again.

This cycle of execution of ***statement2***, evaluation of *test-condition* and execution of ***statement-block*** is continued until the value of *test-condition* becomes ***false***. Once the value of *test-condition* becomes ***false***, the control comes out of the ***for*** loop and it will pass on to the statement that immediately follows the ***for*** statement.

The function of the **for** statement is pictorially illustrated below:

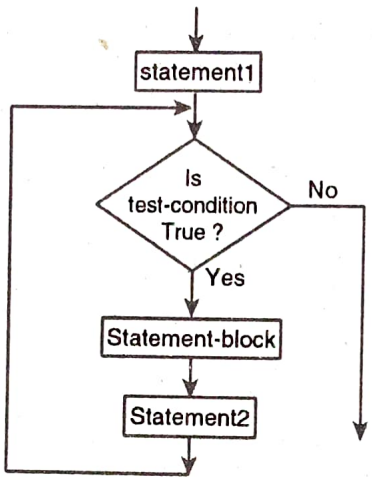

Fig 3.3 *'for' Statement*

As per the above-mentioned function of the ***for*** statement, the following ***for*** statement will result in the display of the "Hello" message 5 times:

```
int  j;
for ( j = 1 ;  j <= 5 ; j++ )
{
    System.out.println( "Hello" ) ;
}
```

Since the body of the above ***for*** statement contains only one statement, we shall write this statement without the braces, as shown below:

```
int  j;
for ( j = 1 ; j <= 5 ; j++ )
    System.out.println( "Hello" ) ;
```

In the above *for* loop, the variable j is known as the **loop counter variable**. We can also declare the loop counter variable inside the *for* loop header, as shown below:

```
for ( int j = 1; j <= 5; j++ )
    System.out.println ( "HELLO" )
```

When we declare the loop counter variable inside the *for* loop header, its use is restricted to the body of the *for* loop. In such a case, the variable j is no longer defined after the *for* loop ends.

As in the case of the ***while*** loop, the ***break*** statement shall be used to bring the control out of the ***for*** loop. When the ***continue*** statement is encountered in a ***for*** loop, control will jump straight to the ***condition***. The following program illustrates the use of ***break*** statement inside a ***for*** loop.

Example 3.5 A positive whole number is first read in. A ***for*** loop is used to extract the factors of the given number. As we know, if there are no factors in the range 2 to (n–1), then that number should be a **prime number**. A ***break*** statement is used to transfer the control out of the ***for*** loop, if there is a factor.

```
// This program is to find the given number prime or not
import java.io.*;
class Prime
{
  public static void main ( String[ ] args ) throws IOException
   {
    InputStreamReader reader = new InputStreamReader ( System.in ) ;
    BufferedReader in = new BufferedReader ( reader ) ;
    System.out.print( " Enter a whole Number : " ) ;
    String text = in.readLine ( ) ;
    int n = Integer.parseInt ( text ) ;
    boolean flag = false ;
    for ( int i = 2 ; i <= n-1 ; i++ )
      if ( n % i == 0 )
      {
        flag = true ;
        break ;
      }
      if ( flag )
        System.out.println( " The Given Number is not a Prime Number " );
      else
        System.out.println( " The Given Number is a Prime Number " );
   }
}
```

Output

```
Enter a whole Number : 7
The Given Number is a Prime Number
```

TRY YOURSELF

I. Objective Type Questions

1. Which one of the following is false in the case of 'if' statement?
   a) This statement is used when there are two possible alternative actions.
   b) The *test condition* may be a logical expression or an arithmetic expression.
   c) The *else* part is optional.
   d) More than one statement may be present in the *then* and *else* parts.
2. Which one of the following is not correct in the case of 'switch' statement?
   a) The *expression* involved in a *switch* statement may be an arithmetic expression.

Chapter 5

Classes and Objects

INTRODUCTION

All the programs that we have discussed in the previous chapters have dealt with numbers and strings. But many real life programs deal with more complex data items, known as *objects*, which closely represent entities in the real world. A bank account, an inventory record or an employee record are simple examples for objects. Java is ideally suited for designing and using objects. In Java, we design classes that describe the behaviour of objects. In this chapter, we will learn how to define and make use of classes and objects.

GENERAL FORM OF A CLASS

A class contains data members and the methods that act upon those data members. The simple general form of a class is as follows:

```
class className
{
    type 1  dataName1;
    type 2  dataName2;
        :
        :
    typeN    dataNameN;
    returnType1   methodName1 ( argumentList1 )
    {
        :
        :
    }
    returnType2    methodName2 ( argumentList2 )
    {
        :
        :
    }
    returnTypeM    methodNameM  ( argumentsListM )
    {
       :
       :
    }
}
```

Here, the keyword **class** indicates that a class named **className** is being declared. The name of a class should follow the usual naming conventions for **identifiers**.

In the above general form, dataName1, dataName2,...,dataNameN are the data members involved in the class. They are *variables*. They are usually declared at the beginning of the class definition, as shown above. The data members are also known as fields.

The declaration of the **methods** follow the declaration of data members as indicated in the above general form. The declaration statement of each **method** will have three parts, namely the **return type**, **method name** and the optional **argument list**.

A large set of commonly used classes, such as *System* and *Math*, have been defined by the Java developers themselves. All such predefined classes in the Java libraries start with an uppercase letter. Objects and methods start with a lowercase letter (such as *out* and *println*). This use of uppercase and lowercase letters is a standard convention of the Java language. The authors of the Java class libraries have followed this standard convention consistently. We should do the same when we define our own classes, objects and methods in our programs.

In the following example, a class named **Rectangle** involves two data members **length** and **width** and two methods **rectangleArea( )** and **rectanglePerimeter( )**.

```
class Rectangle
{
    int length;
    int width;
    int rectangleArea( )
    {
        return length * width ;
    }
    int rectanglePerimeter( )
    {
        return 2 * ( length + width ) ;
    }
}
```

Class is Java's smallest unit of abstraction. Therefore, a variable that is declared inside a class can be accessed by its simple name by all the methods defined in that ***class***. Similarly, a method that is defined in a class shall be invoked by any other method in that class.

For example, in the ***Rectangle*** class defined above, the variables ***length*** and ***width*** are directly referred and used in the **rectangleArea( )**and **rectanglePerimeter( )** methods.

Similarly, the ***rectangleArea( )*** or the ***rectanglePerimeter( )*** method can be directly referred and used in some other method that is present in the ***Rectangle*** class. In the following example, the ***rectangleArea( )*** method is referred and used in the **whiteWashingCost( )** method, which computes the whitewashing cost at the rate of Rs. 15 per square feet.

```
class Rectangle
{
    int   length ;
    int   width ;

    int  rectangleArea(  )
    {
        return   length  *  width ;
```

```
    }
    int  rectanglePerimeter( )
    {
        return  2 * ( length  +  width ) ;
    }
    int  whiteWashingCost( )
    {
        return  rectangleArea( ) * 15 ;
    }
}
```

CREATION OF OBJECTS

A **class** shall be viewed as a sort of template for an **object**. An ***object*** is a single instance of a class that retains the structure and behaviour of the class.

Class is a user-defined data type. We can declare variables of this user-defined data type just as we declare basic type variables. Such variables are called *objects* of the class or *instances* of the class. The process of creating a new object for a class is known as **instantiation**. Once we define a class, we can create any number of objects of its type. The objects of a class are also known as the **instances** of that class.

We have defined a class named ***Rectangle*** in the previous section. We can create a **new** object for this class by using the **new** operator, as shown below:

```
Rectangle  r1 = new Rectangle ( ) ;
```

The *new* operator causes the creation of an object of type *Rectangle* and returns a reference to that object. After the above statement is executed, there will be a new object that has the structure and behaviour as described by the class ***Rectangle***. Here, the variable **r1** is called an **object variable**. It stores the location of the new object. An object location is also known as an **object reference**. When a variable contains the location of an object, we say that it *refers* to that object. In the above example, r1 refers to the *Rectangle* object that has been newly created. It is to be carefully noted here that **r1** does not contain the new object. It refers to the new object.

In Java, every value is either a primitive data type value or an object reference. Familiar primitive data types are int, double, char and boolean. There is an important difference between primitive types and object references in Java. While the primitive type variables can hold values, object references cannot hold objects — they hold only references to objects.

If we wish to compare two object references with the help of the == operator, we test whether the two references refer to the same *object*.

In Java, when an object is passed as an argument to a method, the object is not passed by reference. Actually, the *object reference* is copied by value in such a case.

A Java method can change the values of instance variables involved in an object. But it cannot *replace* the contents of an object reference, as an object reference is passed by value in Java.

Once we define a class, we can create any number of objects of that class. For example, we can create two new objects for the ***Rectangle*** class, by using the following two statements:

```
Rectangle r1 = new Rectangle( ) ;
Rectangle r2 = new Rectangle( ) ;
```

Let us now consider the following two statements:

```
Rectangle r3 = new  Rectangle(  ) ;
Rectangle r4 = r3 ;
```

Here, both the object references **r3** and **r4** will point to the same object. Therefore, any change that we make to the object referenced by **r4** will also affect the same object which **r3** is referring. We can create multiple references to the same object, as illustrated in this example.

Let us now consider the following three statements:

```
Rectangle r5 = new Rectangle(  );
Rectangle r6 = r5;
r5 = null;
```

168193 5179 MUT

Here, the first statement will create a new object and store its reference in **r5**. The same object reference is stored in **r6** by the second statement. Even though the object reference **r5** is set to **null** by the third statement, the object reference **r6** will still point to the object that has been created by the first statement.

Let us now create an object for the class ***Rectangle*** with the help of the following statement:

```
Rectangle r7 = new Rectangle( );
```

Here, the object referenced by **r7** will have its own copy of the variables ***length*** and ***width***.

We can now assign the numeric values 50 and 10 to the variables ***length*** and ***width*** through the following two simple assignment statements:

```
r7.length = 50;
r7.width = 10;
```

Let us now create one more object for the class ***Rectangle***, with the help of the following statement:

```
Rectangle r8 = new Rectangle( );
```

Here, **r8** holds a reference to the newly created object. Here, the object referenced by **r8** will have its own copy of the variables ***length*** and ***width***. We can now assign the numeric values 10 and 5 to the variables ***length*** and ***width*** with the help of the following two sample statements:

```
r8.length = 10;
r8.width  = 5;
```

A variable whose values may vary from object to object is known as an **instance variable**. In the above example, the values of the variables ***length*** and ***width*** in the object **r7** are **50** and **10**. The values of these two variables in the object **r8** are **10** and **5**. Thus, the values of these two variables differ from object to object. Therefore, these two values are ***instance variables***.

An instance variable that is associated with an object of a class is referred by using the dot(.) operator in the following form:

```
objectReferenceVariable.InstanceVariableName;
```

Recall that in the above example, we have referred to the variables ***length*** and ***width*** in the object **r7**, as shown below:

```
r7.length = 50;
r7.width  = 10;
```

Recall further that we have referred to the variables ***length*** and ***width*** in the object **r8**, as shown below:

```
r8.length = 10;
r8.width  = 5;
```

Instance variables are generally declared with the access control specifier *private*. In such a case they can be accessed only by the methods of the same class.

Once we provide the values for the ***length*** and ***width*** in the object referenced by r7 as specified above, we shall use them to compute the area and perimeter of the rectangle by invoking the **rectangleArea( )** and **rectanglePerimeter( )** methods.

A method whose behaviour varies from object to object is known as an **instance method**. Since the values of the variables *length* and *width* of the object **r7** are **50** and **10**, the ***rectangleArea( )*** method will provide the value **500** in this case. Since the values of the variables **length** and **width** of the object **r8** are **10** and **5**, The ***rectangleArea( )*** method will provide the value **50** in the case of the object **r8**.

Since the behaviour of the ***rectangleArea( )*** method differs from object to object, it is an ***instance method***. Similarly, the ***rectanglePerimeter( )*** method in the ***Rectangle*** class is also an ***instance method***.

An instance method that is associated with an object of a class is referred by using the dot(.) operator in the following form:

```
objectReferenceVariable.InstanceMethodName;
```

For example, we can invoke the ***rectangleArea( )*** and ***rectanglePerimeter( )*** methods in the object **r7**, as shown below:

```
r7.rectangleArea( );
r7.rectanglePerimeter( );
```

Similarly, we can invoke the ***rectangleArea( )*** and ***rectanglePerimeter( )*** methods in the object **r8**, as shown below:

```
r8.rectangleArea( );
r8.rectanglePerimeter( );
```

Instance methods are usually declared with the access control specifier *public*. In such a case, they can be accessed by all other methods in the program. Recall that the instance variables are generally declared with the access control specifier *private*. In such a case, all data access occurs through the *public* methods.

USAGE OF CONSTRUCTORS

Java provides a special construct named **constructor** exclusively for creating an object of a class and initializing its instance variables. The name of the ***constructor*** method should be always same as the name of the class. It is generally declared as *public*. It may have an optional list of arguments. It does not have a **return** type, not even **void**. The constructors are not methods. We cannot invoke a constructor on an existing object. We can use a constructor only in combination with the *new* operator.

The constructor for the **Rectangle** class, which was defined in the previous section, shall be described as follows:

```
Rectangle( int a, int b )
{
    length =  a ;
    width  =  b ;
}
```

It shall be carefully noted here that the name of this constructor is same as the name of the class ***Rectangle***. It shall also be noted that it does not have a **return** type.

In most of the cases, the declaration of a constructor will involve a parameter list. The parameters in this list should be always individually declared. For example, the following declaration of the ***Rectangle*** constructor is wrong.

```
Rectangle ( int a, b)
{
    :
    :
}
```

The above-mentioned constructor ***Rectangle*** shall be invoked as shown below:

```
Rectangle  r9   =   new Rectangle( 50,10 );
```

Here, the constructor ***Rectangle*** will be invoked and the values 50 and 10 will be passed as the values of the arguments **a** and **b**. As a result, a new object of the class ***Rectangle*** will be created and the reference to this new object will be stored in the variable **r9**.

If the ***constructor*** is not explicitly declared in a class, then the Java compiler will automatically generate and execute a constructor that has no arguments. Such a constructor is known as the **default** constructor.

When the Java compiler generates and executes a default constructor, it will initialize the instance variables to their default values. While **boolean** type variables will be set to **false**, the numeric type variables will be set to **zero**. Any variable name that acts as a reference to an object of a class will be set to **null**.

Example 5.1 In the following example, the class Rectangle is defined as explained above. Subsequently, two objects of this class are created and then the methods rectangleArea( ) and rectanglePerimeter( ) are invoked.

The following program contains two classes. In general, a program shall contain any number of classes. But only one of them can have the main( ) method. Program name should be same as the name of the class that contains the main( ) method. So, the following program should be saved as **RectangleDemo1.java**

```
class Rectangle
{
int  length;
int  width;
Rectangle ( int a , int b )
{
    length  =  a ;
    width   =  b ;
}
int rectangleArea( )
{
    int  area;
    area = length * width;
    return area;
}
int rectanglePerimeter( )
 {
    int  perimeter;
    perimeter = 2 * ( length + width );
```

```
        return perimeter;
    }
}
class Rectangledemo1
{
    public static void main( String[ ] args )
    {
    Rectangle r1 = new Rectangle(10 , 5);
    System.out.println("Area of Rectangle = " + r1.rectangleArea( ));
    System.out.println("Perimeter of Rectangle = " + r1.rectanglePerimeter( ));
    Rectangle r2 = new Rectangle(7 , 4);
    System.out.println("Area of Rectangle      = " + r2.rectangleArea( ));
    System.out.println("Perimeter of Rectangle = " + r2.rectanglePerimeter( ));
    }
}
```

Output

```
Area of Rectangle      =  50
Perimeter of Rectangle =  30
Area of Rectangle      =  28
Perimeter of Rectangle =  22
```

'this' KEYWORD

The **this** keyword refers to the current object of the class in which it is used.

We shall redefine the ***Rectangle*** constructor defined in Example 5.1, as shown below, by using the ***this*** keyword.

```
Rectangle( int length, int width )
{
    this. length =  length ;
    this. width =  width ;
}
```

Here, **this.length** and **this.width** refer to the variables in the object that is currently created.

It shall be noted here that the unnecessary usage of the additional data names **a** and **b** is avoided here by using the ***this*** keyword. Due to this distinct advantage, the ***this*** keyword is often used in the ***constructor*** method.

CONSTRUCTOR OVERLOADING

The ***Rectangle*** class that has been defined in the previous example has only one constructor. However, a class shall have several constructors, provided they have different **signatures**. As we know, the signature of a constructor is a combination of its name and the sequence of its parameter types. The names of all the constructors of a class should be same as the name of that class. Therefore the overloaded constructors should necessarily have different sequences of parameter types. The **constructor overloading** feature is a good example of **polymorphism**. The following example illustrates ***constructor overloading***.

Example 5.2 In the following example, the class ***Rectangle*** has two overloaded constructors. While the first constructor shall be used when **width** is a constant, the second constructor shall be used when both **length** and **width** vary.

Within the **main( )** method of the ***RectangleDemo2*** class, a new object is created and its reference is stored in **r1** with the help of the first constructor, while another object is created and its reference is stored in **r2** with the help of the second constructor.

```
//Program to illustrate constructor overloading
class Rectangle
{   int  length;
    int  width;
    Rectangle ( int length , int width )
    {
          this.length = length ;
          this.width = width ;
    }
    Rectangle ( int length )
    {
        this.length = length ;
        this.width = 5 ;
    }
    int rectangleArea( )
    {
        int area;
        area = length * width;
        return area;
    }
    int rectanglePerimeter( )
    {
        int perimeter;
        perimeter = 2 * (length + width);
        return perimeter;
    }
}
class Rectangledemo2
{
    public static void main(String[ ] args)
    {
        Rectangle r1 = new Rectangle( 10 , 5);
        System.out.println("Area of Rectangle       = " + r1.rectangleArea( ));
        System.out.println("Perimeter of Rectangle  = " +
                                          r1.rectanglePerimeter( ));
        Rectangle r2 = new Rectangle( 10 );
        System.out.println("Area of Rectangle       = " + r2.rectangleArea( ));
        System.out.println("Perimeter of Rectangle  = " +
                    r2.rectanglePerimeter( ));
    }
}
```

Output

```
Area of Rectangle          =   50
Perimeter of Rectangle     =   30
Area of Rectangle          =   50
Perimeter of Rectangle     =   30
```

COPY CONSTRUCTORS

Sometimes, we may wish to prepare a duplicate of an existing object of a class. In such situations, we have to use a special type of constructor, known as **copy constructor**. This constructor can take only one parameter, which is a reference to an object of the same class.

Example 5.4 In the example given below, the class **Distance** contains two variables **x** and **y**, a ***constructor***, a **copy *constructor*** and the method **distanceFromOrigin( )** to compute the distance between the point(x, y) from the origin. Within the **main( )** method of the **DistanceDemo** class, the object **p** of class ***Distance*** is created by using the usual ***constructor***. One more object of a class ***Distance*** is created by using **copy *constructor***, which takes **p** as its only argument.

```
//Program to illustrate the definition and use of copy constructor
class Distance
{
        int  x,y;
Distance( int  x,  int y )
        {
            this.x = x;
this.y = y;
}
        Distance( Distance p )
{
        this.x = p.x;
this.y = p.y;
}
double distanceFromOrigin( )
        {
return Math.sqrt(  x * x  +  y * y ) ;
}
}
class DistanceDemo
{  public static void main( String[ ] args)
{
        Distance p = new Distance(4,3 );
        System.out.println( p.distanceFromOrigin( ) ) ;
        Distance q = new Distance( p );
        System.out.println( q.distanceFromOrigin( ) ) ;
}
}
```

When we pass the Copy Constructor (Class Object as ar then that is Called Constructor)

Concatination

Output

```
5.0
5.0
```

STATIC DATA MEMBERS

A static variable is defined for a class itself. Its value does not vary from object to object. There will be only one copy of a static variable and all objects of the class will share it. Due to this reason, static variables are known as **class variables**.

A static variable shall be declared by preceding its usual declaration with the **static** keyword, as shown below:

```
static dataType    variableName1;
```

Here are two examples:

```
static boolean flag;
static int count;
```

Several static variables of the same type shall be declared on the same line, as follows:

```
static dataType variableName1,variableName2,.....,variableNameN;
```

Like instance variables, static variables are usually declared with the access specifier *private*. This is done to ensure that methods of other classes cannot change their values.

We ourselves can assign a value to a static variable at the time of its declaration, as follows:

```
static type variableName1 = expression1;
```

Here is an example:

```
static int count = 0;
```

When we don't initialize the static variables, they will be initialized to default values, While **boolean** type variables will be set to **false**, the numeric type variables will be set to **zero**. Any variable name that acts as a reference to an object of a class will be set to **null**.

Sometimes, we may wish to assign initial values to a set of static variables when a class is loaded into memory. In such situations, we shall define and use a block of statements in the form of a **static initialization block**, as shown below:

```
class B
{
static int[ ]    x;
static
    {    x = new init[5];
         for (int i = 0; i < 5; i++)
         x[i] = i;
    }
}
```

The above static initialization block will be executed when the class **B** is loaded into memory. As a result, the values for the five elements of the ~~array~~ **x** will be set to 0,1,2,3, and 4.

Since the static variables are associated with the class itself, they are also known as **class variables**.

A **static variable** can be referred by using the **dot(.)** operator in one of the following two forms:

```
className.staticVariableName;
objectName.staticVariableName;
```

STATIC METHODS

Just like a static variable, a static method is defined for the class itself. Therefore, it is not necessary to create an object of a class in order to invoke a static method defined in it. Such a method shall be directly invoked in the following form:

```
className.staticMethodName( arguments ) ;
```

However, a static method can also be invoked by using an **object**, as shown below:

```
objectName.staticMethodName(arguments);
```

Since the static methods are associated with the class itself, they are also known as **class methods.** When we wish to have a method that is of general utility but does not directly affect the individual instances of a class, we shall declare it as a **static** method.

A static method shall be declared by using the ***static*** keyword as a modifier, as shown below:

```
static   returntype   methodname ( parameters )
{
    .
    .
}
```

A static method can invoke only other static methods. Similarly, they can access only static data members. They cannot refer to the keywords *this* and *super* in any way. We will learn about the keyword *super* in the next chapter.

Example 5.4 In this example, few static variables are declared. The initial values are provided to some of these static variables during the declaration. The other static variables are initialized to the default values.

```
//Program to illustrate value assignment to static variables
class Sample
{
   static int i ; static int j  =  2 ; static int k ;
   static String s1 ; static String s2  =  "Welcome" ;
   static boolean flag;
}
class SampleDemo
{
   public static void main(String[ ] args)
   {
      System.out.println(Sample.i);
      System.out.println(Sample.j);
      System.out.println(Sample.k);
      System.out.println(Sample.s1);
```

```
        System.out.println(Sample.s2);
        System.out.println(Sample.flag);
    }
}
```

Output

```
0
2
0
null
Welcome
false
```

Example 5.5 In the following example,a static variable named **count** and an instance variable named **message** are declared in the class T.

Within the **main( )** method of the **TDemo** class, two objects **t1** and **t2** of the class **T** are created. With respect to each one of these objects, the static variable **count** and the instance variable **message** are accessed.

Since ***count*** is a static variable, one copy of this variable is shared by all the objects of the class ***T***. On the other hand, a separate copy of the instance variable ***message*** exists for each object of the class ***T***.

```
// Program to illustrate the use of static and instance variables
class T
{
    String message;
    static int count = 0;
    T(String message)
    {
        this.message = message;
        count++;
    }
}
class TDemo
{   public static void main(String[ ] args)
        {
            T t1 = new T("Good Morning");
            System.out.println(t1.message + "\t" + t1.count);
            T t2 = new T("Good Afternoon");
            System.out.println(t2.message + "\t" + t2.count);
        }
}
```

Output

```
Good Morning    1
Good Afternoon  2
```

Example 5.6 In the following example, a static initialization block is defined in the class **B**. This block will be executed when the class ***B*** is loaded into memory. As a result the values of the five elements of the array **x** will be set to 0,1,2,3 and 4.

Within the **main( )** method of the class ***B***, the values of the elements of the array x are displayed.

```
// Program to illustrate a use of static initialization block
class B
{  static int[ ] x;
   static
   {   x = new int[5];
       for ( int i = 0; i<5 ; i++)
              x[i] =i;
   }
}
class BDemo
{
   public static void main(String[ ] args)
   {
       for( int i = 0; i< 5; i++)
           System.out.println(B.x[i]);
   }
}
```

Output

```
0
1
2
3
4
```

'finalize( )' METHOD

The Java run–time provides a special utility, known as "garbage collecting system". This system checks the code quite often and removes those objects which are no more referred by the code. But this mechanism does not remove the non-object resources, such as database connections and file descriptors, that are held by an object. We must write code in the *finalize( )* method for removing such non- object resources.

We normally write the *finalize( )* method in a class if it holds one or more non–object resources. The Java run-time automatically invokes the finalize( ) method in a class just before removing an object of that class. Thus, the *finalize( )* method ensures that the non-object resources that are held by an object are removed from the memory before the object itself is removed from the memory.

INNER CLASSES AND ANONYMOUS INNER CLASSES

An **inner class** is one which is defined inside another class. The methods in an inner class have access to the data members and methods in the class within which it is defined.

An inner class is visible only to the class in which it is defined. It is not visible to other parts of the program.