Chapter 1

INTRODUCTION

In this chapter, basic concepts of object-oriented programming (OOP), features of Java language and Java language architecture are explained.

1.1 Object-Oriented Programming Concepts

The object-oriented technique is based on three basic concepts. They are:

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Many computer languages which came earlier, like FORTRAN, BASIC, Pascal, C, etc., handled data in an 'open' manner. Every component (like subroutines, functions) in the program written in those languages can access data defined for the whole program. This feature, though advantageous in many occasions, has some disadvantages in few cases. When a data is declared for all components of a program, any component in that program can alter the data. But, there are problems in which a data is to be protected from modification by all components of the program, barring a particular component.

Not only data, but also the procedures that manipulate the data are to be guarded against misuse by other components of a program. Each procedure or method defined for a specific task is allowed to be accessed only by a particular component or by all other components of a program in varying

degrees. This mechanism of providing protection to data and methods of a program is called encapsulation.

In Java language, encapsulation is realized through a description concept called class. A Java class contains variables representing data and methods that manipulate the data. The methods describe the way in which the data is manipulated. The variables and methods of a class are called members of the class. The members of a class can be declared as private or public. Private members are accessible only within the class. Public members can be accessed both internally and from external components. Fig.1.1 gives a view of a Java class.

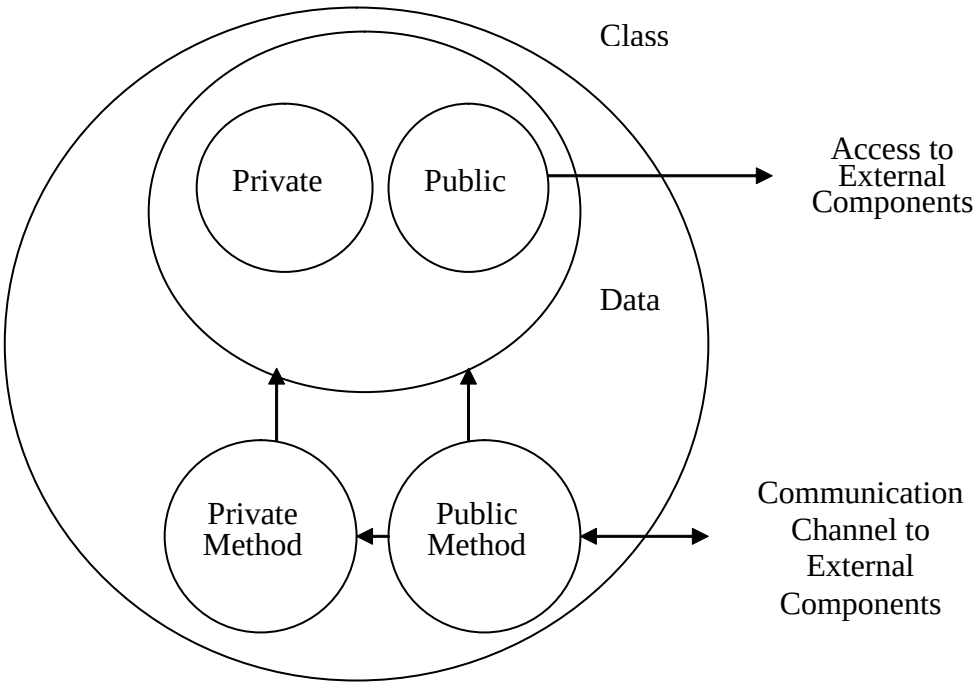

Fig.1.1 A Conceptual View of a Java Class

Public data and methods can be accessed by other components (objects) of a program. The private data and method can be accessed only by that class members. This mechanism provides protection to private members. The only way to access a private member by an external component is through the public method, which is well defined (by the user). Thus public methods encapsulate the data and method and act as an interface. The public method provides a channel for communication with external components.

A class itself cannot be used as such. Realistic entities, called objects, are to be created as per the description of the class, like buildings are constructed using blueprint. Only objects constitute a Java program. Public methods provide communication among objects through which messages are exchanged between objects.

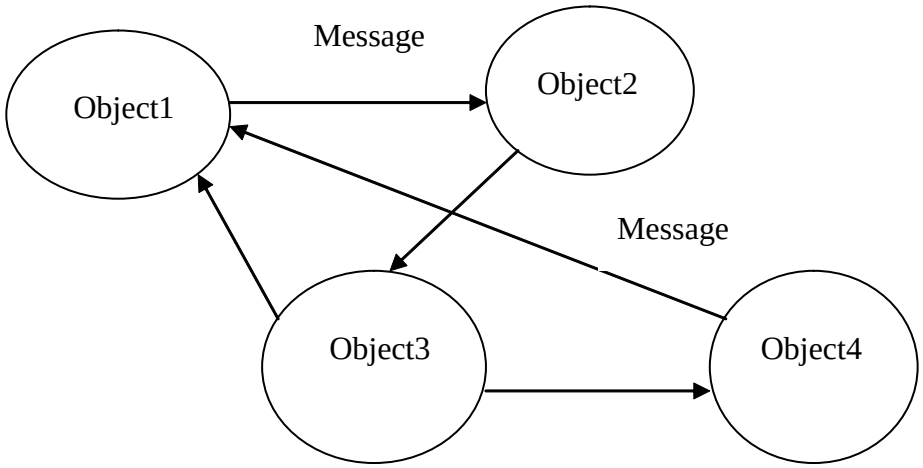

Fig.1.2 Objects Constitute a Java Program

Inheritance

A program module is developed for a particular task. There may be another task, which needs all functionalities of the previous module and a few additional capabilities. In such a case, generally, a new module is developed. Just for a few additional functionalities, one spends time on writing a new module, in which most of the functionalities of the previous module is repeated. Inheritance is used to avoid this repetition. The properties of a module prepared for a task can be inherited in another module and only the additional capabilities required are to be added in that to perform the new task. For example, if there is a module which finds the area of a rectangular surface, then this functionality can be inherited in another module to calculate the volume of a rectangular box. The new module, calculating the volume, need not repeat the procedure for calculating the area. This module needs to have only the procedure to multiply area and height to find the volume. This inheritance property helps to build reusable codes to build complex modules.

In Java, a class is created for carrying out a task. Properties of such classes can be inherited into another class. The process of inheritance can be continued down to any level. Java also provides a mechanism to extract common properties of several classes into a single class, one level above the current level in the class hierarchy. Such classes may not contain exact details, but may only give the broad concept. Such classes are called abstract classes. Fig.1.3 shows the concept of inheritance.

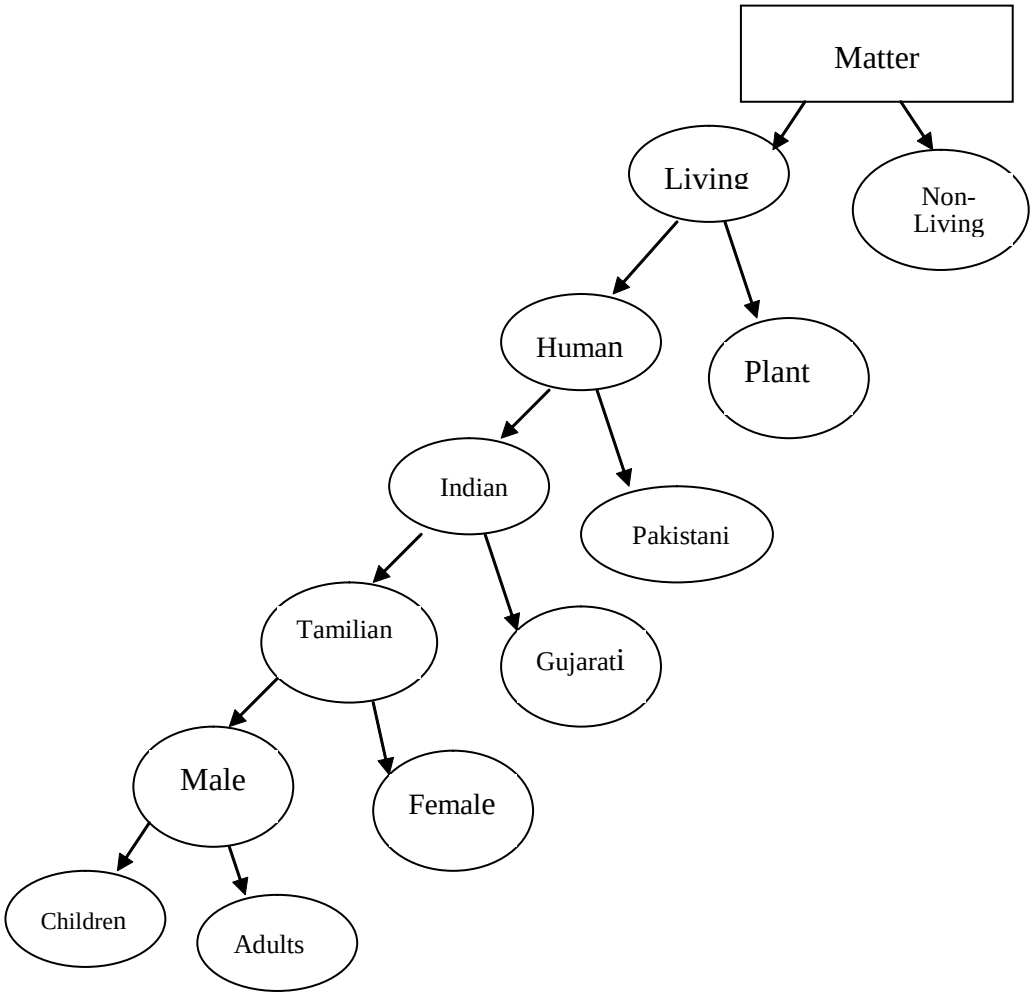

Fig.1.3 The Concept of Inheritance

Matter describes common properties of things present in the universe. The living class gives properties of matter and properties of living thing. The human class has properties of matter plus the properties of living things plus properties of human. In a similar way, inheritance continues down in the inheritance ladder.

Polymorphism

Polymorphism refers to the behavior of the same entity behaving differently in different situations. For example, you feel happy when you receive a good news, feel sad on seeing that you failed in an examination, feel angry when someone teases you and so on. In all the situations, there is only one person, you, but behaving differently.

In Java, one method can be defined in a class, which can perform different tasks depending on the context. Polymorphism in Java is realized through overloading methods and overriding methods.

1.2 Features of Java Language

Java language has some special features, using which programmers can write fast, complex, safe and robust programs. Some of such important features are:

Safe

Java does not provide any pointers like in C or C++. Hence, the memory locations of a system cannot be accessed through a Java program. Therefore, any program developed in Java cannot be used to hack a system.

Robust

Errors that occur at runtime can be handled easily in Java. Java provides exception handling feature to overcome many runtime problems, like divide by zero, memory out of range, input-output, file not found, etc. Using this feature, a user can properly exit or come out smoothly without the program hanging.

Multithreaded

Java language provides an environment by which several tasks can be initiated and managed easily. Such a feature is called multithreading.

Architecture Neutral

A program written and compiled in one platform can run on any other platform running under any type of operating system.

Internet Ready

Java has the capability to handle TCP/IP packets. Hence Java can be used for internet application. It has several classes for internet programming which can be used for client/server programming.

Simple

Many authors of Java have quoted that Java language is simple to learn. No, Java is not that much simple to learn. It is because the capabilities of Java tools are high and one needs to put an effort to understand the Java programming concepts and use them.

1.3 Types of Java Programs

Using Java language, two types of programs can be written. They are:

Application Program

Java can be used for writing programs that run in a PC under the control of the operating system in that machine. Such programs are termed as application programs.

Applet

Programs can be written and compiled to give what is called bytecodes. These bytecodes can be downloaded mostly from a remote server and executed without any control from the local operating system. Such programs are called applets.

1.4 Java Architecture

Java programming environment is based on the following four technologies:

- Java programming language
- Java class files
- Java Application Programming Interface (API)
- Java Virtual Machine (JVM)

The Java source program is created using the features of Java language. The source program is then compiled using the Java compiler, javac, supplied in Java Development Kit (JDK). After compilation, Java class files are created. These class files are in the form of bytecode. These bytecodes can be in the same machine (in application program) or may travel across the network (in applet) and reach the local machine. The Java Virtual Machine executes Java class files and Java API class files as required for Java class files. The API class files have Java native methods that interact with the local operating system. The JVM is a virtual computer developed in software. The combination of JVM and API is called Java platform (see fig.1.4).

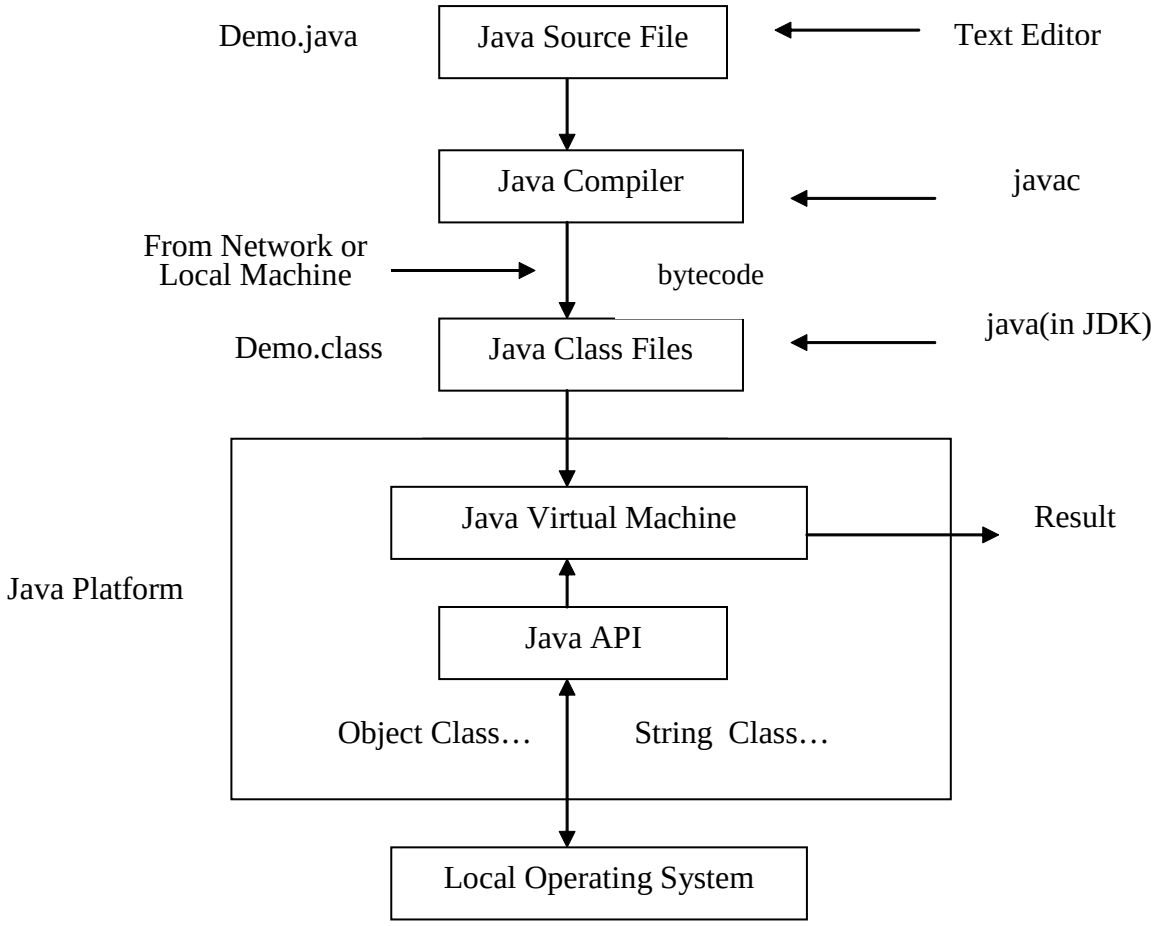

Fig.1.4 Java Architecture

The Java platform is different for different machines. Java platform for Unix, Windows, Macintosh, etc. are different from one another. It is this architecture that makes a Java program to write once, compile once and run in any platform. For a Java program developer, it appears that the program he/she developed can run in any type of platform. The variation in the hardware environment is taken care by different JVMs for different machines, leaving the source program compatible to all types of machines. Hence, a Java program becomes platform independent (see fig.1.5).

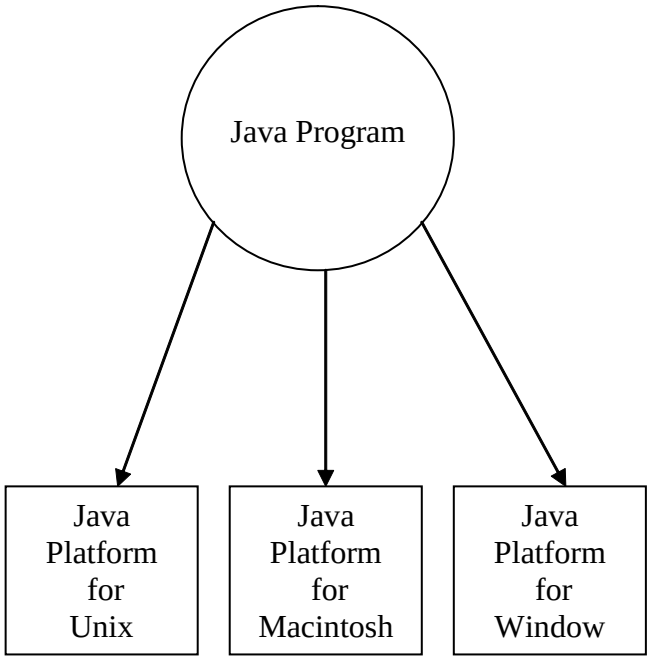

Fig.1.5 Platform-Independent Java Program

After reading this chapter, you should have learned the following concepts:.

- ➲ Object-oriented programming is based on three concepts: encapsulation, inheritance and polymorphism.
- ➲ Objects consitute a Java program.
- ➲ Java language is safe, robust, internet ready and architecture neutral.
- ➲ Java supports two types of programs: applet and application.
- ➲ Java Virtual Machine (JVM) is an abstract computer built using software.

In the next chapter, you will learn about Java literals, data types and variables.

Exercise-1

Fill in the blanks

1.1 The basic principles which form the object-oriented programming are _____, _____ and _____ .

1.2 The mechanism of providing required protection to the components of a program is known as _____ .

1.3 The concept of encapsulation is described in the _____ .

1.4 Java objects are derived from the _____ .

1.5 The absence of pointers in Java helps to write _____ programs that do not hack the system.

1.6 A Java program can be run in any computer running under any operating system. (True/False)

1.7 An application program is run in a computer under control of the _____ in that computer.

1.8 _____ program runs without the control of the _____ .

1.9 Java Virtual Machine and _____ form the Java platform.

1.10 When a Java program is compiled, it gives _____ files.

1.11 Java programs are platform _____ .

1.12 Java Virtual Machine is a virtual computer built using _____ .

* * * * * *

BLANK

Chapter 2

LITERALS, DATA TYPES AND VARIABLES

In this chapter, Java constants called literals, data types and variables are explained.

2.1 Literals

Entities that do not change their values in a program are called constants or literals. While solving real-life problems, one comes across different types of literals. Java provides different literal representations for them.

2.1.1 Integer literals

A whole number is called integer. For example, 25, 75, 399, etc. are integers. Java supports three types of integer literals: decimal, octal and hexadecimal. The above integer numbers are examples of decimal integer literal. Octal integer literals start with 0 (zero) and are followed by octal digits 0 to 7. For example, 0, 037, 02374 are octal integer literals. Hexadecimal integer literals start with 0x (or 0X) and are followed by hexadecimal digits 0 to 9, A to F (a to f). For example, 0X29, 0X3A7, 0X2AB9 are hexadecimal integer literals. Java also provides integers of large magnitude called long integers. To denote such long integers, the letter L (or l) is appended to the literals. For example 732999456789425L and 07777652374237425174325 67L represent long decimal integer literal and long octal integer literal respectively.

2.1.2 Floating Point Literals

Numbers with decimal point and fractional values are called floating point literals. These literals are represented in two forms, standard and scientific. In standard notation, integer part and fractional part are separated by a decimal point. For example, 75.23749 is in standard notation. In scientific notation, a floating point number is followed by the letter E (or e) and a signed integer. The representations 6.237E-35, 4.792E18 and 42e+138 are in scientific notation standing for 6.237×10^{-35}, 4.792×10^{18} and 42×10^{138} respectively.

The accuracy of numbers used in a program depends on the storage capacity inside the computer to store that number. Two standard precisions are used in Java, **float** and **double**. 'float' is called single precision and 'double' is a double precision. By default, all floating point literals are stored in double precision. In order to specify the precision, floating point literals can be appended with f (or F) for **float** and d (or D) for **double.** For example, 2.342f denotes that the number be stored in single-precision **float** and 7.2345678902d or 7.234568902D denotes that the number be stored in double-precision **double**.

2.1.3 Character Literals

Single characters in Java are called character literals. In Java, characters belong to 16-bit character set called Unicode. Java character literals are written within a pair of single quote. For example 'a', 'z' represent character literals. There are certain characters that cannot be printed. Further, in certain occasions, a character like single quote (') itself is to be written as character literal. To represent such characters, Java provides a set of character literals called escape sequence. Table 2.1 gives the escape sequence.

Table 2.1 The Escape Sequence Characters

Escape Sequence	Description
\ddd	Octal character represented in ddd
\Uxxxx	Hexadecimal Unicode character
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Tab
\b	Backspace

2.1.4 String Literals

A sequence of characters written within a pair of double quote is called string literal.
For example:

"This is a String"
"You can put anything here"
"1234 * special characters"
"next \n line"
"\" within double quote\""

are all string literals. Notice that the escape sequence characters are written as such. String literals are to be started and ended in one line only.

2.1.5 Boolean Literals

In Java, boolean literals take two values, **true** or **false**. These two values are not related to any numeric value as in C or C++. The Boolean value **true** is not equal to 1 and **false** is not equal to 0.

2.2 Data Types

While solving real-life problems, one comes across a variety of data types. Therefore, any computer language should be able to handle such data types. Java supports eight basic types. The eight basic types fall under the following four groups:

Integers

Integers are whole valued signed numbers. There are four types in this. They are **byte**, **short, int** and **long**.

Floating Point Numbers

Real numbers, with fractional values and with decimal point, are covered under this category. They are covered under two types, **float** and **double**.

Characters

Single characters are represented through **char** type.

Boolean

Boolean logical values are handled under the type **boolean**.

2.2.1 Integer Types

Integers in Java are handled in four basic types: **byte, short**, **int** and **long**. All types are signed and take positive and negative values. The width of each type is defined by Java language and do not depend on the machine in which the program is executed. The width of each type is prefixed and is platform independent. This is one of the important aspects of Java language in handling integer numbers.

byte

byte is the lowest size integer. **byte** is signed and is 8-bit width. It's value range from -128 to 127. **byte** variables are declared by the keyword **byte**.

Example : byte a, buff;

short

short type is 16-bit width and is signed. It takes the value form -32,768 to 32,767. The **short** type integers are declared by the keyword **short**.

Example : short sn, num;

int

The **int** type is a signed integer of 32-bit width. It takes up the value from -2147483648 to 2147483647. It is the most commonly used type in Java programs. The **int** type integers are declared by the keyword **int**.

Example : int n, numb;

long

The **long** type is a signed integer of 64-bit width. It takes values from -9223372036854775808 to 9223372036854775807. The **long** type integers are declared by the key word **long**.

Example : long factorial, star_count;

The range of integer values for each integer type is defined by Java language and does not depend on the computer on which the numbers are generated.

2.2.2 Floating Point Types

Numbers with fractional values are called floating point numbers and are known as real numbers in older languages like FORTRAN. The floating point numbers are represented in two forms, **float** and **double.**

float

In **float** type, the numbers are specified in 32-bit width. It takes value from 3.43e-038 to 3.43e+038. This is a single precision. Single-precision numbers are processed efficiently. The **float** type variables are declared by the keyword **float**.

Example: float x, area;

By default, all floating point numbers are treated as **double**. Hence, explicit specification is required when initializing **float** variables. To represent a float, the number is to be appended with f.

Example: float x = 85.25f;

double

The **double** type floating point numbers are represented in 64-bit width. This is a double-precision representation. It takes value from 1.7e-308 to 1.7e+308. By default, all floating point numbers assume this form. It is the most commonly used form. The **double** type variables are declared by the keyword **double**.

Example: double volume, average;

2.2.3 Character Type

Single characters are handled by **char** type. It is a 16-bit code. Therefore, it can represent 65,536 distinct characters. In Java, this 16-bit representation of characters is called **Unicode**. It covers a large set of language characters in the world. For example, characters of English, Gujarati, Telugu, Tamil and Hindi languages are defined in the Unicode character set in addition to several other languages. The **char** type variables are declared by the key word **char**.

Example: char choice, flag;

The **char** type is associated with ASCII value, which is a subset of unicode value. Therefore, **char** type can also be associated with **int** values. For example, character 'A' is represented by the value 65.

Characters defined in Java are 16-bit width and are called Unicode characters.

2.2.4 Boolean Type

The logical values **true** and **false** are handled by **boolean** type. The boolean values **true** and **false** are not associated with any numerical value. (In C or C++, zero is treated as false and non-zero value as true). The **boolean** type variables are declared by the keyword **boolean.**

Example: boolean flag, full, empty;

2.3 Variables

Variables, as the name indicates, take different values during the execution of the program. A few authors term variables as an identifier. A variable is any combination of letter, number, underscore and $ sign. The variable must not begin with numbers. A letter may be any Unicode character defined for a language. There is no maximum limit on the total number of characters that form the variable.

A Few Valid Java Variables Are:

sum	total_val	area	fact_num
_max	$value	Total	

A Few Invalid Java Variables Are:

2nd_class	min-val	length&breadth
avg val	total/	float
double		

float and double are Java-reserved words and hence invalid.

Java variables are case-sensitive, i.e., uppercase letters are different from those of lowercase. Therefore, the variables, sum and Sum, are different. All Java variables required for the program must be declared before it is used and it can be placed anywhere in the block where they are used.

Declaring A Variable

A variable is declared with a type it is going to store. An initial value can also be assigned along with the declaration. The general form of declaring a variable is :

type variable [=value], variable [=value];

Example :

```
int x, y, z;
int a, b = 45, c;
float p, q = 7.253f;
char c, choice = 'y';
```

Scope of Variables

The scope refers to the validity of a variable across the Java program. The scope of a variable is limited to the block defined within the braces { and }. It means a variable cannot be accessed outside the scope.

Default Values for Basic Types

When variables are declared in a program, they need to be assigned some value before they are used in that program. When no value is assigned after declaration, the basic types assume certain default initial values. Table 2.2 shows the default values that variables take.

Table 2.2 Default Initial Values for Different Types

Type	Initial Value
char	null
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
boolean	false

The scope of variables is restricted to the block in which it is defined.

After reading this chapter, you should have learned the following concepts:

- Constants defined in Java are called literals.
- Characters defined in Java are called Unicode characters.
- Java defines eight basic types:

  byte, short, char, int, long, float, double and boolean
- The range of integer values for each integer type is defined by Java language and does not depend on the computer in which the numbers are generated.

In the next chapter, you will learn the structure and syntax of a Java program.

Exercise-2

I. Fill in the blanks

2.1. Constants are called as _____ in Java language.

2.2. A hexadecimal literal starts with _____ .

2.3. 035 is a/an _____ integer.

2.4. In Java language the precision of a number _____ on the machine in which the program is executed.

2.5. There are _____ basic types in Java.

2.6. _____ type in Java is called Unicode character.

2.7. The scope of a variable is restricted to _____ in which they are declared.

2.8. In the following list _____ are invalid Java variables:

Sum
_Max
$value
1class
Mat-value
float

2.9. For engineering and scientific calculations _____ type will be more useful.

2.10. Non-printable characters are represented by _____ characters.

2.11. The default initial value for char type is _____ .

2.12. The value 75.25 as such is to be assigned to a _____ type variable.

* * * * * *

3.2 Comments

Comments can be inserted into a Java program in three different forms. In the first form, comments can be written in a single line by placing // at any location of a statement. All statements made after // and till the end of the line will be treated as comment.

Example:

i) // This is a first Java program
ii) a = p * Q ; // making a multiplication
iii) int x, y ; //declaring two int types.

In the second form, comments running into multiple lines can be made. In this form, comments start with /* and end with */. The beginning /* and ending */ can be in the same or different lines.

Example:

i)
```
/*_____________________________________________
This program is developed to find a prime number
_____________________________________________*/
```

ii)
```
/*
This is an example for multi line comment
*/
```

The third form of comment is meant for generating an HTML file that documents your program. This form of comments starts with /** and ends with */.

3.3 Expressions and Statements

Java expressions, like in any other language, consist of variables and literals separated by operators. A Java expression will be of the form:

An example for an expression is :

40 * m + total / numb - 25

The operands in the expression may be the same or different type. If the operands are of the same type, the resulting value of the expression is also the same type. Suppose, if all variables and literals are **int** type, the resulting value is **int** type.

When a Java expression is assigned to a variable, it becomes a statement. The general form of a statement is:

Variable = expression ;

3.4 Type Conversion

It is important to know beforehand what type an expression will give when it is evaluated. It is necessary because an expression may contain different types. Java has mechanisms to handle different types present in an expression during evaluation or while assignment. Type conversions of primitive types occur while evaluating an expression or during assignment. Java does the conversion through two mechanisms, automatic promotion and type casting.

Automatic Promotion

Java automatically converts variables or literals of lower precision type to a higher precision type during the evaluation of expression or during assignment. This is known as automatic promotion or widening. When unary operators like -,+,--,++ etc. operate on an operand of type byte or short, the operand will be converted to int type, otherwise the operands are left as such. When binary operations take place, the type of the operands are checked. If one of the operands is double, the other will be converted to double. If one of the operands is float, the other will be converted to float and if one of the operands is long, the other operand will be converted to long, otherwise the two operands will be converted to int (if the operands are of type byte or short).

Example:

i)
```
byte b1 = 35;
byte b2  = –b1;
```

The second statement will give compile error because b1 is automatically promoted to int.

ii)
```
byte b1 = 25;
byte b2 = 14;
byte b3 = b1+b2;
```

The third statement will give compile error as both b1 and b2 will be promoted to int before the binary operator + is operated. Hence, the addition will give an int result, whereas the left hand side b3 is a byte.

iii)
```
byte b1 = 25;
byte b2 = 14;
int n = b1+b2;
```

The third statement is valid and correct.

Java also does automatic conversion while assigning values to variables in a statement:

destination = source;

where the source may be an expression or variable or even a literal. In such assignments, if precision of the destination is larger than the source, Java automatically converts the lower precision value (say int) to higher precision value (say long). Table 3.1 shows the source type and destination type.

Table 3.1 Valid Type Conversion

Source Type	Destination Type
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Fig. 3.1 shows the widening type conversion.

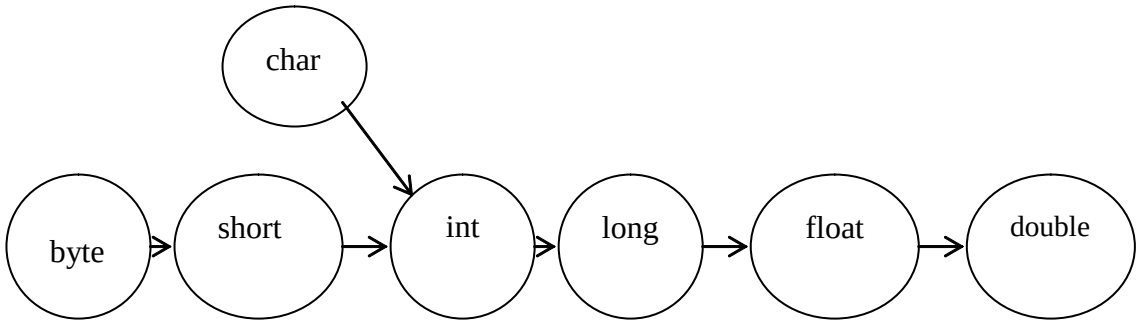

Fig.3.1 Widening Type Conversion

Widening can take left to right as indicated in the arrow mark in fig.3.1 in consecutive stages or hop one or more stages in between. For example, **byte** can directly be converted to **long** type.

byte and short type are automatically converted to int in an expression. Hence, the destination type should be declared according to this conversion.

Type Casting

There are situations where incompatible type conversion is needed, where a higher precision type is being assigned to or converted to a lower precision. For example, it may be needed to get an **int** type from a **double** type or a **byte** from a **float**. This process is known as narrowing or down casting. The narrowing process will result in loss of bit and the value obtained may be incorrect. Therefore, a programmer must know the consequence of narrowing before effecting it. The type conversion is to be done explicitly by using the keyword **type**. The general form of type casting is:

type variable = (type) expression;

Whenever incompatible type conversion is needed, type casting is used.

3.5 Block Statements and Scope

Statements written in single lines are simple statements. There are occasions, where a group of statements are to be treated as one unit. In such situations, several simple statements are enclosed between a pair of braces { and } and is called block statement. Blocks define the scope of variables. As we have seen earlier, the scope of variables is restricted to the blocks in which they are declared. Blocks can be nested. The following program 3.3 illustrates the use of nested block statement:

Program 3.2

```
class BlockDemo
    {
    public static void main(String args [])
        {
        int num = 24;

        double y;
        {
        int i = 45, k;
        k = num + i;
        System.out.println("k = " + k);
        }
        y = num / 3.0;
        System.out.println("y = " + y);
        }
    }
```

The above program gives the following output:

```
k = 69
y = 8.0
```

The variables i and k are declared inside the inner block. Hence, their scope is valid only in that block. Outside the block, i and k cannot be accessed. The variables num and y are declared in the outer block and, therefore, can be accessed in both blocks. Consider the following program 3.3:

Program 3.3

```
// This program illustrates the scope of variables
// in nested blocks.
//  somasundaramk@yahoo.com
class Scope
    {
    public static void main(String args [])
        { // main block
        int num = 25;
        double y;
        { // second block
        int i = 45;
        int k;
        k = num + i;      // num can be accessed here
        System.out.println(" K = " + k);
        }                 // second block ends
        y = k / 3.0;      // k is outside its scope. hence
                          //   compile error will occur
        System.out.println(" Y = " + y);
        }                 // main block ends
    }
```

The statement y=k/3.0 in program 3.3 attempts to access k outside its scope and hence will produce compile error.

Though Java restricts the scope of a variable to the corresponding block, variables with identical name can not be declared in two blocks. For this reason, the following statements will give compile error:

```
{     int i ;
.
.
.
   { int i ;                 // i is defined already
                             // will give error
.
.
.
```

```
    }
}
```

The following program 3.4 illustrates the concepts discussed in this chapter:

Program 3.4

```
/* This program illustrates the following concepts:
    Single-line comment
    multi-line comment
    statement block
    type casting
*/
class Program33
    {
    public static void main(String args [])
        {
        int n = 2785, i = 15, ix;
        double x = 76.89, y, z, mix;
        byte bn;
        {
        int m = 25, k;
        k = m + i;        // i can be accessed here also
        System.out.println("k = " + k);
        }                          // scope of k and m ends
        y = n / 26.85;
        // z = k/2.5;           // do not try this statement
        bn = (byte)n;          // converting int to byte
        ix = (int)x;           // converting double to int
        System.out.println("y = " + y);
        System.out.println("n = " + n + "   byte n = " + bn);
        System.out.println("x = " + x + "   int x = " + ix);
        }
    }
```

The above program gives the following output:

```
k = 40
y = 103.570100409074
n = 2785     byte n = –31
x = 76.89    int x = 76
```

After reading this chapter, you should have learned the following concepts:

- ➲ The structure of a Java program
- ➲ Making comments in a Java program
- ➲ Scope of variables
- ➲ Automatic promotion of byte and short to int
- ➲ Type casting

In the next chapter, you will learn about the various operators defined in Java language.

Exercise-3

I. Fill in the blanks

3.1. A class is defined within a pair _____ .

3.2. Each statement in a Java program is terminated with _____ .

3.3. A Java program must contain a _____ method.

3.4. Comments in Java program can be made in _____ ways.

3.5. Java automatically converts byte and short to _____ .

3.6. When incompatible type conversion is needed _____ is used.

3.7. A set of statements enclosed between { and } is called _____ statement.

3.8. The scope and life of a variable is restricted to the _____ in which they are declared.

3.9. The file name of a Java program must be the same as that of the _____ name.

3.10. Variable names defined in different blocks can _____ the same .

3.11. An expression with operands of type byte and float will give a result of _____ type.

II. Write Java programs for the following:

3.12. Write a program that prints your address.

3.13. Write a program to find the average of three byte type numbers.

3.14. Write a program to find the sum of five short type numbers.

* * * * * *

Blank

Chapter 4

OPERATORS

In this chapter, various operators used in Java language are explained.

Operators operate on operands and cause changes in the operand value or give a new value. Java provides operators in four categories. They are:

1. Arithmetic operators
2. Bitwise operators
3. Relational operators
4. Logical operators

Operators may operate on one, two or three operands. Operators operating on single operand are called unary operators, on two operands are binary operators and on three operands are ternary operators. Unary operators are further classified as prefix operator and postfix operator. When the operator precedes the operand, it is called prefix operator and when it follows, it is called postfix operator.

4.1 Arithmetic Operators

Arithmetic operators operate on arithmetic variable and arithmetic literals. The arithmetic operators and the character symbol that represent them are listed in table 4.1.

Table 4.1 Arithmetic Operators

Operation	Operator Symbol
Add	+
Subtract	–
Multiply	*
Divide	/
Modulus	%

The first three operators carry the same meaning as we do in normal calculations. The division operator gives integer division if both operands are integer and floating point division, if one of the operands is a floating point.

For example:

18/4 gives 4
18/4.0 gives 4.5

Modulus Operator

The modulus operator % gives remainder value after a division. It works for both integer and floating point value.

For example:

25 % 5 will give 0
25 % 7 will give 4
25 % –7 will give 4
–25 % 7 will give –4
25.8 % 7 will give 4.8
25.8 % 7.0 will give 4.8
25.8 % 7.5 will give 3.3 (quotient is integer)

Modulus operator percentage works for both integer type and floating point values.

All the five arithmetic operators are binary operators, i.e. they need two operands. However, the subtract operator can also be used as unary prefix operator as given below:

```
y = –x;
```

The same is applicable for + operator and can be written as:

```
p = + q;
```

This form, though valid and correct, is not generally used, as the statement

```
p = q;
```

means the same.

Operator Assignment

All the five arithmetic operators can be used in operator assignment form. This helps to avoid repeating the operands in two places of a statement. Consider the statement:

```
sum = sum + 1;
```

This statement can also be written as:

```
sum += 1;
```

This form of assigning value to the variable is called operator assignment. In general, a statement of the form:

variable = variable operator expression

is replaced by operator assignment form:

variable operator= expression;

For arithmetic operators, the following forms +=, –=, *=, /= and %= are applicable.

Increment and Decrement Operators

There are two more arithmetic operators ++ and ––. Both are unary operators and operate only on integers. The ++ increment operator increases the operand's value by 1. Both can operate in prefix form ++n or in postfix form n++, where n is a variable. In prefix form, the value of the operand n is incremented first and the operand is used. In postfix form, the value of the operand is used first and then incremented. The same is applicable to the decrement operator ––. Program 4.1 illustrates the use of increment and decrement operators.

The prefix operator (++variable or ––variable) changes (increment by one or decrement by one) the value of the variable first and uses the variable. When postfix operator (variable++ or variable––) is used, the value of the variable is used in the operation and then changes the value (increment by one or decrement by one).

Program 4.1

```
// This program  illustrates the use of increment and
// decrement operators.
class InrDcr
    {
    public static void main(String args [])
```

```
    {
    int m = 25, p = 70;
    System.out.println("m = " + m);
    System.out.println("++m = " + ++m);
    System.out.println("p = " + p);
    System.out.println("p++ = " + p++);
    System.out.println("p++ after use = " + p);
   }
}
```

The above program gives the following output:

```
m = 25
++m = 26
p = 70
p++ = 70
p++ after use = 71
```

The ++ and –– operators do not operate on integer literal. Therefore, 25++ or ––85 are not valid.

The ++ and –– operators should not be operated on integer literals (like ++75).

4.2 Bitwise Operators

Bitwise operators are used to manipulate individual bits of a data item. There are situations where individual bits of a data are to be modified. Java provides a set of bitwise operators. These operators operate only on **byte**, **char**, **short**, **int** and **long** types. The bitwise operators in Java are given in table 4.2.

Table 4.2 Bitwise Operators

Operation	Operator Symbol
Bitwise NOT	~
Bitwise AND	&
Bitwise OR	|
Bitwise exclusive OR	^
Left shift	<<
Right shift	>>
Right shift zero fill	>>>

Bitwise NOT (~)

This is a unary operator. It complements each bit of the operand. Consider a number 71. This number in byte type has a binary value of 01000111.

a = 01000111

The NOT operation gives:

~a = 10111000
= –96

Bitwise AND (&)

This AND operator, & , performs AND operation bit by bit of the operands. Consider two operands a = 71 and b = 25, represented in byte type as:

a = 01000111
b = 00011001

The AND operation gives:

a & b = 00000001
= 1

Bitwise OR (|)

This OR operator, | , performs OR operation bit by bit. Consider two bytes a = 71 and b = 25. The OR operation gives:

a | b = 01011111
= 95

Bitwise Exclusive OR (^)

This XOR operator, ^ , performs exclusive OR (XOR) bit by bit. In this operation, if the corresponding bits of the operands are identical, the resulting bit is 0, otherwise it is 1. For the numbers a = 71 and b = 25, the XOR operation gives:

a ^ b = 01011110
= 94

Left Shift Operator (<<)

This operator, << , is a unary operator. It shifts each bit of the operand to the left by the specified number of positions. The general form of using this operator is:

variable << n

where n specifies the number of positions each bit in the variable is to be shifted left. The leftmost bit is shifted out and the rightmost bit is filled with zero.

Example:

a = 25
= 0 0 0 1 1 0 0 1

The left shift operation,

a << 2

specifies that each bit be shifted left by 2 positions. The result,

after 1st shift is 0 0 1 1 0 0 1 0

after 2nd shift is 0 1 1 0 0 1 0 0

a = 100

Thus, the final value of a is 100. Each left shift is equivalent to multiplication by 2. One has to remember that Java promotes automatically the **byte** and **short** type to **int** type before evaluating them. Hence, care must be taken to handle such situation. When bits in signed integers are shifted left, the leftmost bit indicating the sign will be retained always, i.e. a negative value after the shift will also be a negative number.

Example:

a = –25
= 10011001

The operation a << 2 will give

a = 11100100
= –100

Right Shift Operator (>>)

This operator >> shifts each bit of the operand to the right by a specified number of positions. The rightmost bit is shifted out and lost. The leftmost bit is sign extended. All other bits in between are shifted to right by one position. Each right shift is equivalent to divide by 2. The general form of using this operator is:

variable >> n

where n specifies the number of positions the bits in the variable are to be shifted right.

Example

```
a    =    25
     =    00011001
```

The operation a >> 2 gives

```
a    =    00000110
     =    6
```

For a signed number, the sign bit will be extended. The leftmost bit with 1 indicates that the number is negative.

Example

```
b  =   –25
   =   10011001
```

The operation b >> 2 gives

```
b =    11100110
```

Notice that the leftmost sign bit is sign extended to the right each time and, hence, the two extra 1s at the left.

Here also the problem of automatic promotion of **byte** and **short** type to **int** type is to be handled according to the requirement.

Both right shift (>>) and left shift (<<) operators retain the sign bits even after shifting. Hence, a programmer must be aware of the consequence of this when negative integers are used.

Right Shift Zero Fill Operator (>>>)

This operator, >>>, performs the same type of operation done by right shift operator, >>, except that sign bit extension is not done. Instead, zero will be inserted at the leftmost bit position for every shift. Therefore, in situations where the sign extension is unwanted, the right shift zero fill operator can be used. This operation is also called unsigned shift, as the other two shift operators, >> and <<, extend the sign bit. The general form of using this operator is:

```
variable >>> n
```

where n specifies the number of bit positions of the variable to be shifted right with zero fill.

Example

```
b    =    –25
     =    10011001
```

The operation b >>> 2 gives

```
b    =    00100110
     =    38
```

You may compare this result with the earlier one with >> operator.

The reader is again reminded of the problem of automatic promotion of **byte** and **short** type to **int** type by Java. In most cases, masking off the higher order bytes in the int type will provide a solution to face such situations.

For all bitwise operators, the operator assignment form is also applicable.

Examples

1. a = a>>3
   can be written as
   a >> = 3

2. b = b & c can be written as
   b &= c

3. y = y>>>n can be written as
   y >>>= n

4.3 Relational Operators

For comparing the values of variables and literals, Java provides relational operators. Relational operators are used to relate a given value with several possible values of a variable. The results of the relational operators help to make branching, iterating a block and to terminate the block statement. The relational operators are given in table 4.3.

Table 4.3 Relational Operators

Operation	Operator Symbol
Equal to	==
Not equal to	!=
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

All relational operators are binary operators. The result of operating a relational operator is boolean, **true** or **false**. They operate on operands of type **char**, **byte, short**, **int**, **long**, **float** and **double**. For string variables and literals, separate methods are used and are given in chapter 14.

Examples

```
int a = 25;
int b = 75;
double x = 35.87;
double y = 67.43;
char c1 = 'b';
char c2 = 'd';
```

1. a == b gives false
2. a != b gives true
3. a < b gives true
4. x > y gives false
5. c1 == c2 gives false
6. c1 < c2 gives false
7. x > b gives false
8. 35 == 10 gives false

4.4 Boolean Logical Operators

Logical operators operate only on boolean operands and not on numerical operands. Logical operators result in boolean values **true** or **false**. The boolean logical operators in Java are given in table 4.4.

Table 4.4 Boolean Logical Operators

Operation	Operator Symbol
Logical AND	&
Logical OR	|
Logical XOR	^
Short-circuit OR	||
Short-circuit AND	&&
Logical unary NOT	!
Equal to	==
Not equal to	!=
Ternary if-else	?:

These logical operators, except the unary NOT and ternary if-else, are binary operators. They are used in the same way as that for bitwise operators. The unary NOT complements true to false and false to true.

The following table 4.5 gives results of various logical operations on the boolean variables A and B:

Table 4.5 Truth Table for Boolean Operators

A	B	A|B	A&B	A^B	!A
False	False	False	False	False	True
False	True	True	False	True	True
True	False	True	False	True	False
True	True	True	True	False	False

The following examples illustrate the use of logical operators:

Examples

```
int a = 25;
int b = 75;
double x = 34.25;
double y = 63.98;
char c1 = 'a';
char c2 = 'd';
```

1. (a>b) & (x<y)
   = false & true
   = false
2. (a>b) | (x<y)
   = false | true
   = true
3. (c1 == c2) ^ (b > 100)
   = false ^ false
   = false
4. (c1 != c2) ^ (b > 100)
   = true ^ false
   = true
5. ! (a>b)
   = ! false
   = true
6. (a>b) == (x<y)
   = false == true
   = false
7. (a>b) != (x<y)
   = false != true
   = true
8. (a<b) ? (x>y) : (c1 != c2)
   = true ? false : true
   = false

9. (a>b) ? (x>y) : (c1 != c2)
   = false ? false : true
   = true

Short-circuit && and || operators

From the examples given above, you will find that the expressions on both sides of the logical operators are evaluated and then only the final result is obtained. However, from the first four columns of the Table 4.5 you will notice that :

i) if one of the operands of the OR operation is true, then the result is always true irrespective of the value of the other operand.

ii) if one of the operands of the AND operation is false, the result is always false, irrespective of the value of the other operand.

Therefore, in a logical OR operation, if the first operand has a value true, the result can be declared as true without checking or evaluating the value of the second operand. In a similar way, in an AND operation, if the first operand has a value false, the result can be declared as false without even evaluating the second operand. This feature is built into the short-circuit AND, &&, and short-circuit OR, ||, operators. These are more efficient than & and | as they save time spent for evaluating the second operand. The following examples illustrate the use of && and || operators.

Examples

```
int a      =   25;
int b      =   75;
double x   =   34.25;
double y   =   63.98;
char c1    =   'a';
char c2    =   'd';
```

1) (a>b) && (x<y)
   = false (no need to evaluate the expression x < y)

2) (c1! = c2) || (a<b)
   = true (no need to evaluate a<b)

3) (a>b) || (x<y)
   = false || true
   = true

   only in this kind of situations both operands are evaluated to get the final result.

The short-circuit logical AND (&&) and short-circuit logical OR (||) are more efficient than logical AND (&) and logical OR (|).

4.5 Ternary Operator (?)

The ternary operator, ?, operates on three operands. This operator can replace simple if-then-else statements. The general form of using this operator is:

expression1 ? expression2 : expression3

The expression1 should always result in a boolean value true or false. If the expression1 results in true, the expression2 is evaluated and expression3 is discarded, otherwise, expression3 is evaluated and expression2 is discarded. Program 4.2 illustrates the use of a ternary operator.

Program 4.2

```
// This program illustrates the use of a ternary operator.
class TernaryProg
    {
    public static void main(String args [])
        {
        int mark = 75;
        String result;
        result = (mark > 40) ? “Pass” : “fail”;
        System.out.println(“Your Mark  = “ + mark);
        System.out.println(“Result = “ + result);
       }
    }
```

The above program gives the following output:

```
Your Mark  = 75
Result = Pass
```

4.6 Operator Precedence

An expression contains a number of operators, variables and literals. A programmer must know how these operators are evaluated. Like in other computer languages, priorities are given for each operator. Operators are evaluated on the priorities allotted to them. This is called operator precedence. A good knowledge of these priorities is necessary to write unambiguous Java expressions. In most of the cases, an expression is evaluated from left to right of the expression. Unary and assignment operators take right to left association. The Java operator precedence from highest to lowest is given in table 4.6.

Table 4.6 Operator Precedence

Highest	() []
	! ++ – – +(unary) -(unary) (cast)
	* / %
	+ –
	<< >> >>>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?:
Lowest	= op=

The operators in a row have the same precedence and are evaluated in the order in which they appear in the expression. Use of parentheses in an expression alters the precedence and it is hence listed at the highest precedence or priority.

Example:

```
int a=25, b=60, c=5;
double x = 5.5, y;
y = a+b / (2*c)–x
  = 25 + 60 / (2x5)–5.5
= 25 + 60/10 – 5.5
= 25 + 6 – 5.5
= 31 – 5.5
= 31.0 – 5.5
= 25.5
```

After reading this chapter, you should have learned the following:

- Arithmetic operators
- Bitwise operators
- Relational operators
- Logical operators and short circuit logical operators
- Operator precedence

In the next chapter, you will learn the control statements.

Exercise–4

I. Fill in the blanks

4.1. There are _____ categories of operators in Java.

4.2. The result of the division 7/2 is _____ .

4.3. The result of the operation 7.5%2.0 is _____ .

4.4. The value of y in the following statements is _____ .

```
int x = 50, y;
y = x++;
```

4.5. The following statement is _____ (valid/invalid):

```
y = ++70;
```

4.6. The bitwise operators can operate on float type operands.(True/False)

4.7. The sign independent bitwise operator is _____ .

4.8. The equality operators operate only on integer variables or values. (True/False)

4.9. The result of using logical & and logical && is the same. (True/False)

4.10. The use of parentheses in an expression will alter the precedence. (True/False)

II. Write a Java Program for the Following:

4.11. The register number, name and marks for three subjects are given. Write a program to find the average of the three marks and print it out along with the register number and name.

4.12. A cloth shop during festival seasons offers a discount of 12% on all purchases made in that shop. The bill amount for a customer is given as Rs.750.5. Write a program to calculate the discount, amount after discount and print it out.

4.13. A bank gives 6.5% per annum compound interest on deposits made in that bank. Write a program to calculate the total amount that a person will receive after the end of 4 years for a deposit of Rs.5000/-.

4.14. A motorcycle dealer sells two-wheelers to his customers on loan, which is to be repaid in 3 years. The dealer charges a simple interest of 14% for the whole term on the day of giving the loan itself. The total amount is then divided by 36 (months) and is collected as equated monthly installment (EMI). Write a program to calculate the EMI for a loan of Rs.39990. Print the EMI value in rupees.

* * * * * *