

## UNIT STRUCTURE

- 1.1 Learning Objectives
- 1.2 Introduction
- 1.3 Overview of Java
- 1.4 Basic Features of Java
- 1.5 C/C++ and Java Language Family
- 1.6 Platform Independence of Java
- 1.7 Java Environment
- 1.8 Installing the Java SDK
- 1.9 Creating and Running Java Programs
- 1.10 Let Us Sum Up
- 1.11 Answers to Check Your Progress
- 1.12 Further Readings
- 1.13 Model Questions

---

### 1.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to :

- define combinational circuit
- gain the concept of Java
- differentiate Java as Object Oriented Language
- illustrate the basic features of Java
- know the Java environment
- know how to install Java SDK
- create Java Programs and compile & run them

---

### 1.2 INTRODUCTION

---

The greatest challenges and most exciting opportunities for software developers today lie in tackling the power of networks. Most of the applications created today, will almost certainly be run on machines connected to the global networks i.e. Internet.

We know that by the mid 1990s, the World Wide Web had transformed the online world. Through a system of hypertext, users of the Web

were able to select and view information from all over the world. However, while this system of hypertext gave users a high degree of selectivity over the information they chose to view, their level of interactivity with that information was low. Moreover, the Web lacked true interactivity—real-time, dynamic, and visual interaction between the user and application.

Sun Microsystems, a company best known for its high-end Unix workstations, developed a programming language named **Java** to create software that can run on many different kinds of devices. Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level.

Java brings this missing interactivity to the Web. With a Java-enabled Web browser, you can encounter animations and interactive applications. Java programmers can make customized media formats and information protocols that can be displayed in any Java-enabled browser. Java's features enrich the communication, information, and interaction on the Web by enabling users to distribute executable content—rather than just HTML pages and multimedia files—to users. This ability to distribute executable content is the power of Java.

In this unit, we will introduce you to the Java programming language. We will discuss the basic features of Java and how to install the Java Development Kit. We will also discuss how to write a Java program and the procedure for compiling and running a Java program.

---

## 1.3 OVERVIEW OF JAVA

---

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. Work on Java originally began with the goal of creating a platform-independent language and operating system for consumer electronics. The original intent was to use C++, but as work progressed in this direction, the Java developers realized that they would be better served by creating their own language rather than extending C++. This language was

initially called “Oak” but was renamed “Java” in 1994 as the Web emerged. Then after Java was used as the basis for a Web browser, called WebRunner. WebRunner was successfully demonstrated, and the Java/HotJava project took off.

HotJava, Java, and the Java documentation and source code were made available over the Web, as an alpha version, in early 1995. Initially Java was hosted on SPARC Solaris, and then on Windows NT. In the summer of 1995, Java was ported to Windows 95 and Linux. In the fall of 1995 the Java Beta 1 version was released through Sun's Web site, and Java support was introduced in the Netscape 2.0 browser.

The Java Beta 1 release led scores of vendors to license Java technology, and Java porting efforts were initiated for all major operating systems.

In December 1995 the Java Beta 2 version was released, and JavaScript was announced by Sun and Netscape. Java's success became inevitable when, in early December, both Microsoft and IBM announced their intention to license Java technology.

On January 23, 1996, Java 1.0 was officially released and made available for download over the Internet. JavaScript was also released. Netscape 2.0 now provides support for both Java and JavaScript.

---

## 1.4 BASIC FEATURES OF JAVA

---

The Java team has summed up the basic features of Java with the following list of buzzwords :

- **Simple**

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.



### NOTE

#### **HotJava**

HotJava is a Web browser that is written in Java. HotJava is a Java-enabled browser. This means that HotJava can execute Java applets contained on Web pages. In order to accomplish this, HotJava calls the Java runtime system. The Netscape 2.0 browser, like HotJava, is also Java enabled. It contains a copy of the Java runtime system embedded within it.

- **Object-oriented**

Java is a true object-oriented language. Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions.

- **Robust**

Java is a robust language. The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

- **Secure**

Prior to Java, most users did not download executable programs frequently from Internet, and those who did scanned them for viruses prior to execution. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. Java answers both of these concerns by providing a "firewall" between a networked application and your computer. When you use a Java-compatible Web browser,

you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

- **Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.

- **Architecture-neutral**

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

- **Portable**

In addition to being architecture-neutral, Java code is also portable. It was an important design goal of Java that it be portable so that as new architectures (due to hardware, operating system, or both) are Java and the runtime environment is written in POSIX-compliant C.

- **Distributed**

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra-address space messaging.

This allowed objects on two different computers to execute procedures remotely. Java has recently revived these interfaces in a package called *Remote Method Invocation (RMI)*. This feature brings an unparalleled level of abstraction to client/server programming.

- **Dynamic**

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

---

## 1.5 C/C++ AND JAVA LANGUAGE FAMILY

---

C was developed to meet general system programming needs. It was quickly adapted for general use and found widespread acceptance. C is a high-level procedural language that has many low-level features. These features help to make it versatile and efficient. However, many of these features give it a reputation for being cryptic and hard to maintain. C++ extends the C language to provide object-oriented features. The language is backward compatible with C, and code from the two languages can be used with each other with little difficulty. C++ has found quick acceptance and is supported by a number of pre-built specialized classes.

Java can be considered the third generation of the C/C++ family. It is not backward compatible with C/C++ but was designed to be very similar to these languages. The creators of Java intentionally left out some of the features of C/C++ that have been problematic for programmers. Java is strongly object-oriented. In fact, one cannot create Java code that is not object-oriented. Java's portability is a key advantage and is the reason why Java is often used for Web development.

---

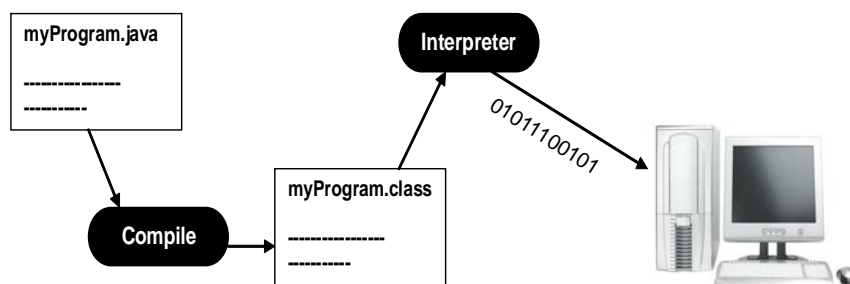
## 1.6 PLATFORM INDEPENDENCE OF JAVA

---

We are already familiar how to compile a C or C++ program. With

most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. It means - **Java combines both these approaches thus making Java a two-stage system.** The Java development environment has two parts : a *Java compiler* and a *Java interpreter*.

In the Java programming language, all source code is first written in plain text files ending with the **.java** extension. Those source files are then compiled into **.class** files by the **javac** compiler. With the compiler, first you translate a program into an inter-mediate code called **Java bytecodes**. Bytecodes are not machine instructions and therefore, in the second stage, **Java interpreter** generates machine code that can be directly executed by the machine that is running the Java program. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed.

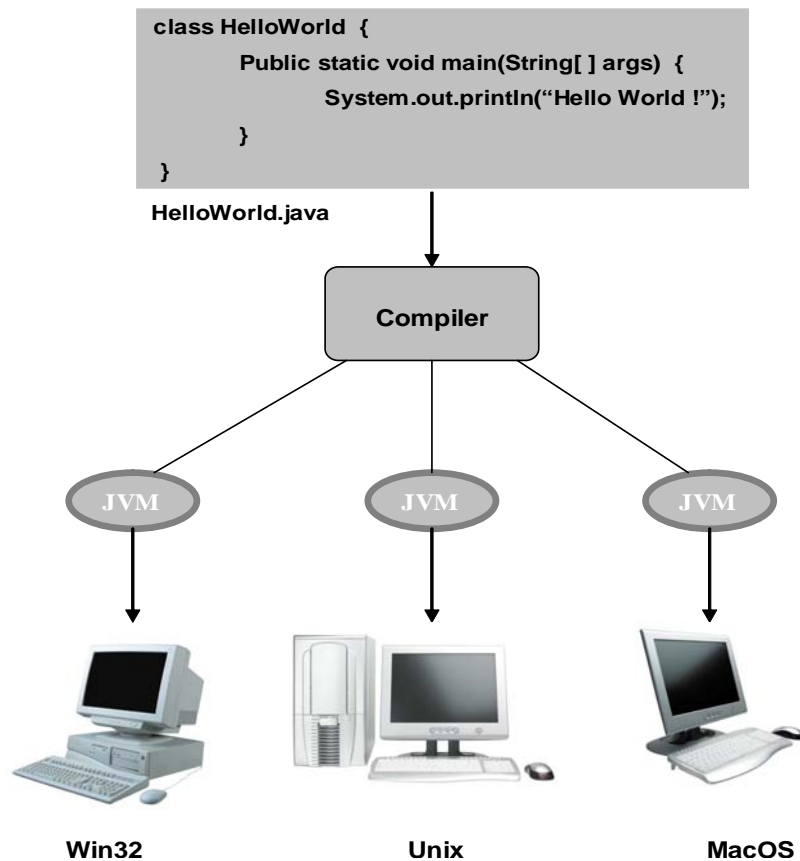


**Fig. 1.1 A Java program first compiled and then interpreted**

Java bytecodes help make “write once, run anywhere” possible. You can compile your program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java VM. That means that as long as a computer has a Java VM, the same program written in the Java programming language can run on Windows 2000, a Unix workstation, or on an iMac, as shown in Fig 1.2

**NOTE****Java Virtual Machine (JVM)**

Java compiler produces an intermediate code known as **bytecode** for a machine that does not exist. This machine is called the **JVM** and it exists only inside the computer memory. The bytecodes are also known as *virtual machine code* which are not the actual machine code. The actual machine codes are generated by the Java interpreter only.



**Fig1.2 Program written once and can run on almost any platform**

Normally, when you compile a program written in C or C++ or in most other languages, the compiler translates your program into machine codes or processor instructions. Those instructions are specific to the processor your computer is running—so, for example, if you compile your code on a Pentium system, the resulting program will run only on other Pentium systems. If you want to use the same program on another system you have to go back to your original source, get a compiler for that system, and recompile your code.

---

## 1.7 JAVA ENVIRONMENT

---

Java environment includes a large number of development tools and hundreds of classes and methods. The development tools are part of



the system known as *Java development Kit* (JDK) and the classes and methods are part of the *Java Standard Library* (JSL), also known as the *Application Programming Interface* (API).

### Java development Kit

The Java development Kit comes with a collection of tools that are used for development and running Java programs. Some of they are :

<i>java</i>	The loader for Java applications. This tool is an interpreter and can interpret the class files generated by the <i>javac</i> compiler.
<i>javac</i>	The compiler, which converts source code into Java bytecode
<i>jar</i>	The archiver, which packages related class libraries into a single JAR file.
<i>javadoc</i>	The documentation generator, which automatically generates documentation from source code comments
<i>jdb</i>	The Java debugger
<i>jps</i>	The process status tool, which displays process information for current Java processes
<i>javap</i>	The class file disassembler
<i>appletviewer</i>	This tool can be used to run and debug Java applets without a web browser.
<i>javah</i>	The C header and stub generator, used to write native methods

An **application programming interface (API)** is an interface

implemented by a software program to enable interaction with other software, similar to the way a user interface facilitates interaction between humans and computers. Java APIs include hundreds of classes and methods grouped into several functional packages. Most commonly used packages are :

- Language support package
- Utility package
- Input/Output Package
- Networking Package
- AWT(Abstract Window Tool Kit) Package
- Applet Package

---

## 1.8 INSTALLING THE JAVA SDK

---

Java is a programming language that allows programs to be written that can then be run on more than one type of operating system. A program written in Java can run on Windows, UNIX, Linux etc. as long as there is a Java runtime environment installed.

For the first time, you just want to run Java programs so, download the Java Runtime Environment, or JRE. Suppose, you want to develop applications for Java, download the Java Development Kit, or JDK. The JDK includes the JRE, so you do not have to download both separately.

You can download the JDK from the Sun Microsystems, free of charge by using this URL <http://java.sun.com/javase/downloads/index.jsp> . The first download page should look like the top page shown in fig 1.3.

http://java.sun.com/javase/downloads/index.jsp

ORACLE® Sun Developer Network (SDN)

Sun Java Solaris Communities My SDN Account

APIs Downloads Products Support Training Participate

SDN Home > Java Technology > Java SE >

## Java SE Downloads

**Download the complete platform and runtime environment**  
Download the Java SE-JavaFX bundle, and use your creative talents to design a winning application. » [Get the bundle](#)

Overview Technologies Documentation Community Support **Downloads**

Latest Release | Next Release (Early Access) | Embedded Use | Real-Time | Previous Releases

**Download**

Java Platform (JDK)  
» [JDK](#) » [JRE](#)

**Download**

JDK + JavaFX Bundle

**Download**

JDK + NetBeans Bundle

**Download**

JDK + Java EE Bundle

Here are the Java SE downloads in detail.

**Java Platform, Standard Edition**

**JDK 6 Update 18 (JDK or JRE)**  
This release includes performance improvements and support for newer releases of these operating systems: Ubuntu, SuSe Linux, and Red Hat Linux. » [Learn more](#)

**Download JDK**

[Docs](#)

**Free Java Download**

- » Check downloads for all operating systems
- » Read more about Supported System Configurations

**Java for Business**

- » Access to critical fixes
- » Long-term support
- » Enterprise features
- » JRE or JDK 6, 5.0, or 1.4.2

**Java Expert?**  
Get paid everytime you answer Java questions!

[Learn More](#)

LIVEPERSON

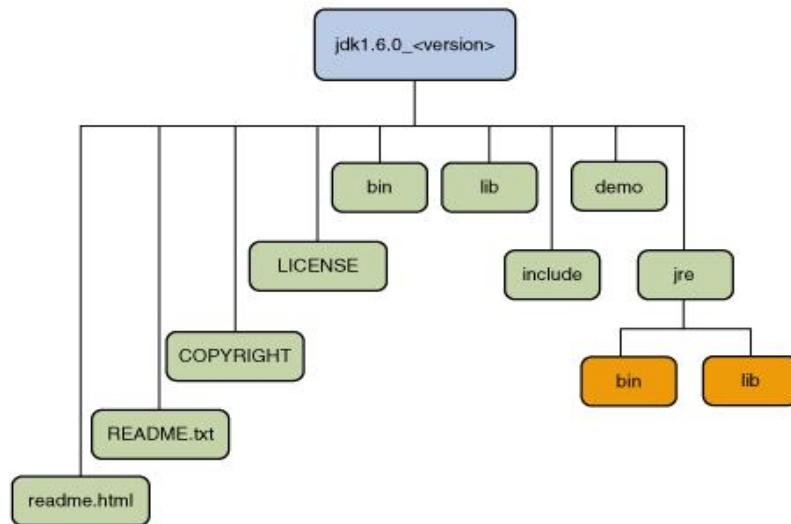
Fig 1.3 Download page for JDK

Click on the **Download JDK** tab shown, and this will save the **jdk-6u18-windows-i586.exe** file on your computer.

After the download is complete, for installing the software, double click the .exe file, and this will automatically install the software by giving some instruction.

The JDK has the directory structure as shown in the fig 1.4.

To compile a Java program on the command line, you will use the command **javac** and to run a compiled program on the command line you will use the **java** command. These two commands are executed by running the **javac.exe** and **java.exe** programs that are located in the **bin** folder of the folder **jdk1.6.0\_18** in C drive. For your operating system (e.g. Windows or Linux) to be able to run these programs, you have to tell it where they are i.e. you can set the PATH variable if you want to be able to conveniently run the JDK executables (javac.exe, java.exe, javadoc.exe, etc.) from any directory.



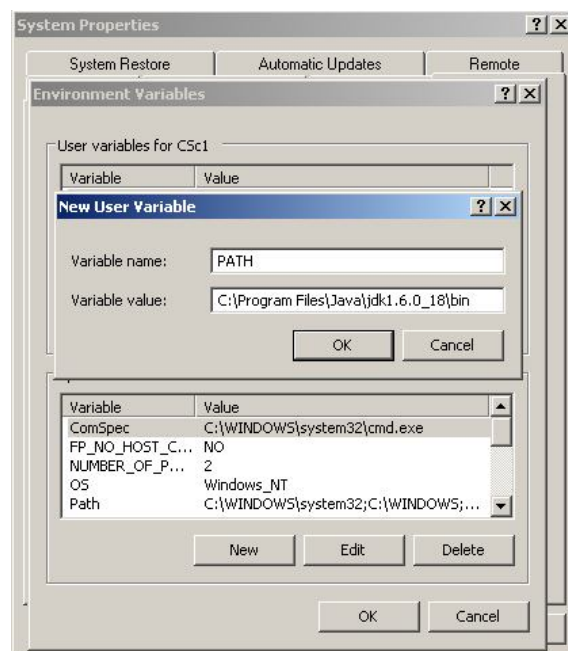
**Fig. 1.4 JDK directory structure**

If you don't set the PATH variable, you need to specify the full path to the executable every time you run it, such as:

```
C:> "C:\Program Files\Java\jdk1.6.0_<version>\bin\javac" MyClass.java
```

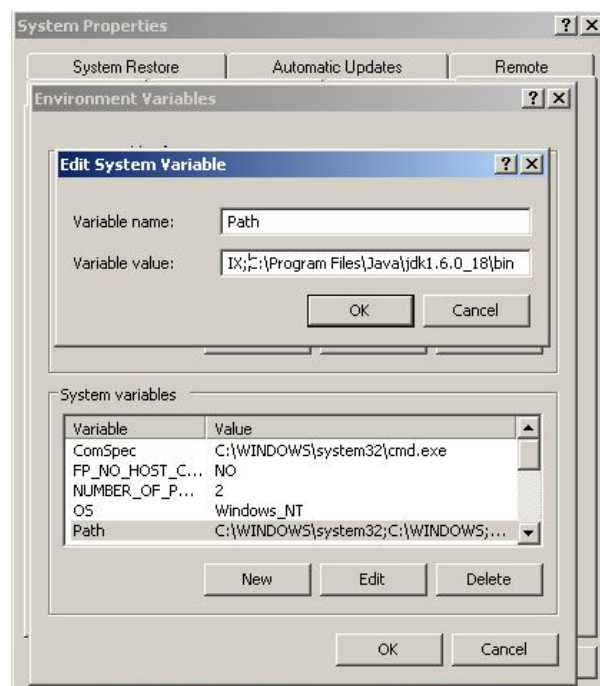
**Procedure for setting the PATH variable :**

- a) Click Start > Control Panel > System on Windows XP or Start > Settings > Control Panel > System on Windows 2000.
- b) Click Advanced > Environment Variables.
- c) Add the location of bin folder of JDK installation for PATH in **User Variables** and **System Variables**.
  - i) For adding User variable Click New and type  
Variable name - PATH  
Variable value - C:\Program Files\Java\jdk1.6.0\_18\bin



**Fig 1.5 Setting the User variable**

- ii) For adding the System variable first select the **path** variable, click on Edit tab and add the above variable value (which we have add for User variable) by giving ; to the existing values. Figure for setting the system variable is shown below :



**Fig. 1.6 Setting the System Variable**

## 1.9 CREATING AND RUNNING JAVA PROGRAMS

---

So far you have learn a few basic about Java and the way of installing the Java SDK. Now, we will concentrate how to write programs using Java programming language. We can develop two types of Java Programs :

- Stand alone applications
- Java applets

Stand alone applications are java programs that can carry out certain tasks on local computer. Java applets are small programs that are used to developed Internet applications. Java applets can be downloaded from a Web server and run on your computer by a Java-compatible Web browser, such as Netscape Navigator or Microsoft Internet Explorer. **We will discuss about the Java applets in next unit.**

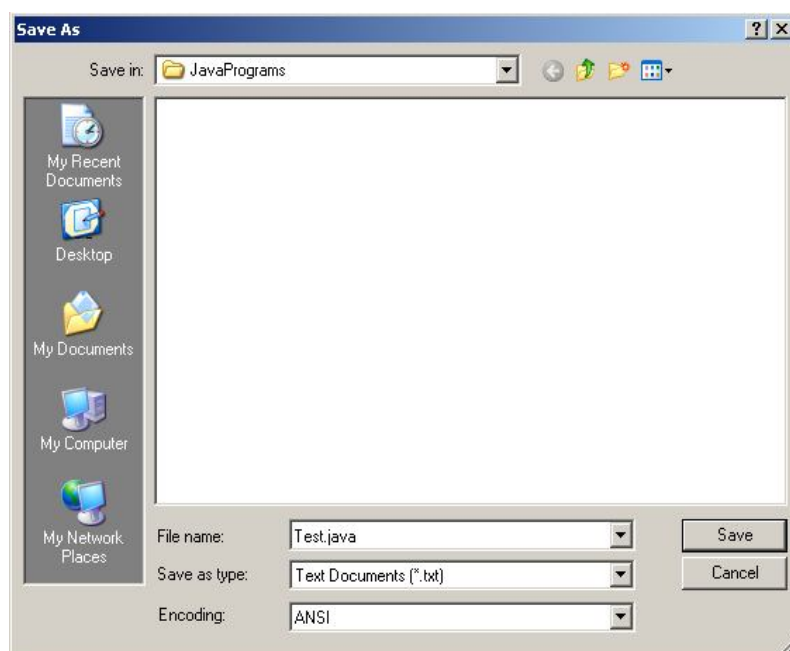
We are already familiar, how to write programs and how to compile and run them. In case of Java also, we will follow the three steps givlen below :

- **Create a source file.** A source file contains text, written in the Java programming language, that you and other programmers can understand. You can use any text editor (e.g. Notepad) to create and to edit source files.
- **Compile the source file into a bytecode file.** The compiler takes your source file and translates the text into instructions that the Java VM can understand. The compiler converts these instructions into a bytecode file.
- **Run the program contained in the bytecode file.** The Java interpreter installed on your computer implements the Java VM. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.

Let us try to write the following simple Java program using Notepad:

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("I am A Simple Program!");
    }
}
```

After typing the program, save it in a folder named as **JavaPrograms** in C drive by giving the file name same as the class name (here it is Test) with the extension **.java**. Here, in our example the name of our file will be **Test.java**. The following figure shows how you will save.



**Fig.1.7 Saving Test.java file**

For compiling the source file, from the **Start** menu select the **Command Prompt**. Change your current directory to the one in which your file is located. For example, we have created the directory **JavaPrograms** in the C drive where we keeps the source files, so we need to change the current directory by typing (press **Enter**) the following command :

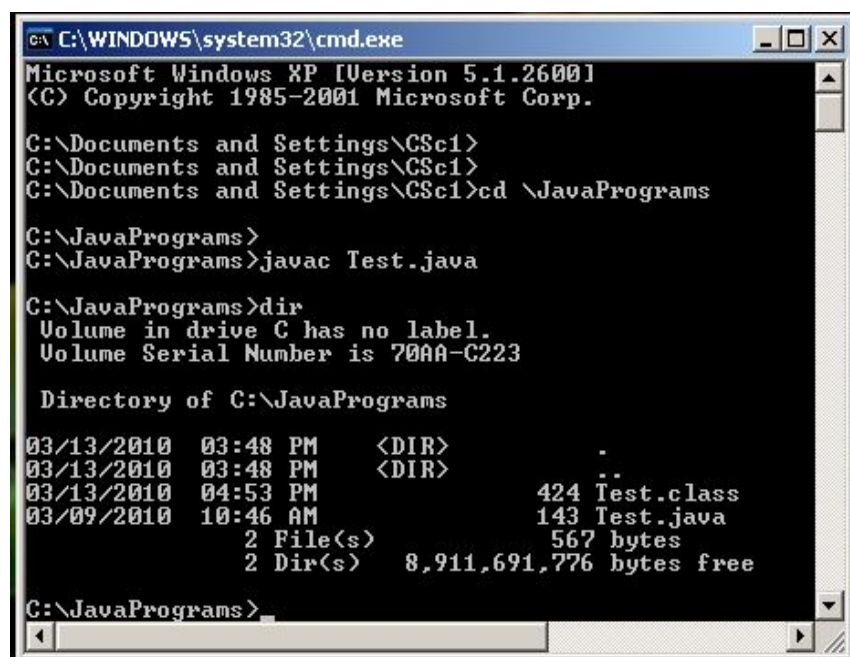
**cd \JavaPrograms**

After entering the above command your Command Prompt will look like the following figure 1.8.

Now, for compiling the source file i.e. **Test.java** enter the following command at Command Prompt -

**javac Test.java**

If your prompt reappears without error messages, then you have successfully compiled your program. It means the compiler has generated a Java bytecode file **Test.class** in the same directory. You can see the **.class** file by typing **dir** command at Command Prompt as shown in the Fig. 1.8.



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\CSc1>
C:\Documents and Settings\CSc1>
C:\Documents and Settings\CSc1>cd \JavaPrograms

C:\JavaPrograms>
C:\JavaPrograms>javac Test.java

C:\JavaPrograms>dir
Volume in drive C has no label.
Volume Serial Number is 70AA-C223

Directory of C:\JavaPrograms

03/13/2010  03:48 PM    <DIR>          .
03/13/2010  03:48 PM    <DIR>          ..
03/13/2010  04:53 PM                424 Test.class
03/09/2010  10:46 AM                143 Test.java
                2 File(s)                567 bytes
                2 Dir(s)  8,911,691,776 bytes free

C:\JavaPrograms>
```

**Fig. 1.8 Compiling a Java Program**

Now, you can run your program by typing the following command -

**java Test**

Fig. 1.9 shows the result what you will see -



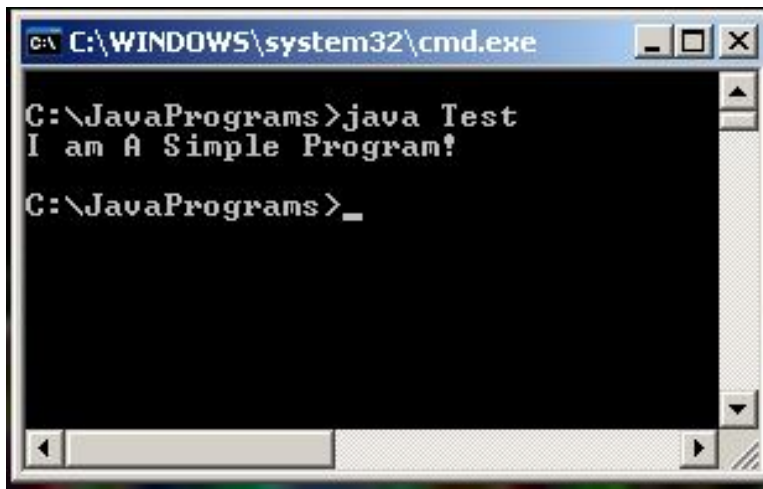


Fig. 1.9 Running a Java Program

### Class Declaration

A class is the basic building block of an object-oriented language, such as the Java programming language. The above Java program consists of a *main class*, named **Test**, which contains a main function, named **main()**. The following bold text begins the class definition block for the application :

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("I am A Simple Program!");
    }
}
```

### The main() Method or Fuction

The following bold text begins the definition of the main method :

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("I am A Simple Program!");
    }
}
```

Every Java program must include the `main()` function declared like this:

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

Conceptually, this is similar to the ***main()*** function in C / C++ and it's the entry point for your application and will subsequently invoke all the other methods required by your program.

The main method declaration starts with three modifiers whose meanings are given below :

**public** : means allows any class to call the main method

**static** : means that the main method is associated with the **Test** class as a whole instead of operating on an instance or object of the class

**void** : indicates that the main method does not return a value

As you can see from the declaration of the ***main()*** function,

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

it accepts a single argument i.e. **String[]** means an array of elements of type *String*. The name of this array is **args** (for "arguments"). This array is the mechanism through which the Java Virtual Machine passes information to your application.

### The Output Line

The only executable statement in the program is

```
System.out.println("I am A Simple Program!");
```

This is similar to the ***"printf()"*** statement of C or ***"cout <<"*** of C++. The **println** method is a member of the **out** object, which is a static data member of System class. This line prints the string -

```
I am A Simple Program!
```

to the screen. The method **println** always appends a newline character to the end of the string. This means that every output will be start on

a new line. Always remember that every java statement must end with a semicolon.



## CHECK YOUR PROGRESS

1. What command invokes the Java compiler from the command line?

.....  
.....  
.....

2. What program usually runs a Java applet ? Name another program that can run a Java applet.

.....  
.....  
.....

3. What is a Java virtual machine ?

.....  
.....  
.....

4. What kind of files contain Java bytecode ?

.....  
.....  
.....

5. What is JDK ?

.....  
.....  
.....

## 1.10 LET US SUM UP

Java is purely object-oriented. Java borrows C++ syntax, but avoids many of C++'s problem areas. Java produces programs that are robust and secure. The Java development environment has two parts : a *Java compiler* and a *Java interpreter*. To compile a Java program on the command line, use the command **javac** and to run a compiled program on the command line use the **java** command. There are two

categories of Java programs : *Java applications and Java applets*. Applets are Java programs that are downloaded and run as part of a Web page. Applets can create animations, games, interactive programs, and other multimedia effects on Web pages. Every Java program must include the main() function declared like this: **public static void main(String[] args).**



## 1.11 ANSWERS TO CHECK YOUR PROGRESS

---

1. javac
2. Browsers usually run Java applets, though several programs (including appletviewer and HotJava) can also run Java applets.
3. A Java virtual machine is a software system that translates and executes Java bytecodes
4. A Java source code file is saved in a file with the extension **.java**
5. JDK stands for Java Development Kit. It describes the set of files that can be downloaded from Sun Microsystems for developing Java applications. It includes the Java compiler and the Java API.



## 1.12 FURTHER READINGS

---

1. Java Programming Language Handbook by Anthony Potts, David H. Friedel Jr. , Coriolis Group Books
2. Programming with Java- A Primer by E Balagurusamy, Tata McGrawHill



## 1.13 MODEL QUESTIONS

---

1. Describe any three basic features of Java programming language.

### UNIT STRUCTURE

- 2.1 Learning Objectives
- 2.2 Introduction
- 2.3 Java Tokens
- 2.4 Variables
  - 2.4.1 The Scope of Variables
- 2.5 Constants
- 2.6 Data Types
- 2.7 Operators and Expressions
- 2.8 Control Flow Statements
  - 2.8.1 Decision Making Statements
  - 2.8.2 Looping
  - 2.8.3 Branching Statements
- 2.9 Let Us Sum Up
- 2.10 Answers to Check Your Progress
- 2.11 Further Readings
- 2.12 Possible Questions

---

### 2.1 LEARNING OBJECTIVES

---

After going through this unit, you will be able to :

- learn about Java tokens
- learn about the variables, constants and data types in Java
- declare and define variables in Java
- learn about the various operators used in Java programming
- describe and use the control flow statements in Java

---

### 2.2 INTRODUCTION

---

The previous unit is an introductory unit where you have acquainted with the object-oriented features of the programming language Java. The installation procedure of Java SDK is also described in the unit. You have learnt how to write, save, compile and execute programs in Java from the previous unit.

In this unit we will discuss the basics of Java programming language which include tokens, variables, data types, constants etc. This might be a review for learners who have learnt the languages like C/C++ earlier. We extend this discussion by adding some new concepts associated with Java. Different control flow statements like if, if-else, while, for, break, continue etc. will also be covered in this unit.

## 2.3 JAVA TOKENS

The smallest individual units in a program are known as **tokens**. A Java program is basically a collection of classes. There are five types of tokens in Java language. They are: **Keywords**, **Identifiers**, **Literals**, **Operators** and **Separators**.

- **Keywords:** Keywords are some reserved words which have some definite meaning. Java language has reserved 60 words as keywords. They cannot be used as variable name and they are written in lower-case letter. Since Java is case-sensitive, one can use these words as identifiers by changing one or more letters to upper-case. But generally it should be avoided. Java does not use many keywords of C/C++ language but it has some new keywords which are not present in C/C++. A list of Java keywords are given in the following table:

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	synchronized	this
threadsasafe	throw	throws	transient	true
try	var	void	volatile	while

**Table 2.1 : Java Keywords**

- **Identifiers** : Java Identifiers are used for naming classes, methods, variables, objects, labels in a program. These are actually tokens designed by programmers. There are a few rules for naming the identifiers. These are:
  - Identifier name may consists of alphabets, digits, dolar (\$), character, underscore(\_).
  - Identifier name must not begin with a digit
  - Upper case and lowercase letters are distinct.
  - Blank space is not allowed in a identifier name.
  - They can be of any length.

While writing Java programs, the following naming conventions should be followed by programmers :

- \* All local and private variables use only lower-case letters. Under-score is combined if required. For example,

total\_marks

average

- \* When more than one word are used in a name, the second and subsequent words are marked with a leading upper-case letter. For example,

dateOfBirth,

totalMarks,

studentName

- \* Names of all public methods and interface variables start with a leading lower-case letters. For example,

total, average

- \* All classes and interfaces start with a leading upper-case letter. For example,

HelloJava

Employee

ComplexNumber

- \* Variables that represent constant values use all upper-case letters and underscore between word if required. For example,

PI

RATE

MAX\_VALUE

- **Literals:** Literals in Java are a sequence of characters such as digits, letters and other characters that represent constant values to be stored in a variable.
- **Operators:** An operator is a symbol that takes one or more arguments and operates on them to produce a result. We will explain various operators in *section 2.7*.
- **Separators:** Separators are symbols used to indicate where groups of code are arranged and divided. Java separators are as follows:

{ }	Braces
( )	Parentheses
[ ]	Brackets
;	Semicolon
,	Comma
.	Period

---

## 2.4 VARIABLES

---

A variable is an identifier that denotes a storage location where a value of data can be stored. The value of a variable in a particular program may change during the execution of the program.

We must provide a name and a type for each variable we want to use in our program. The variable's name must be a legal identifier. The variable's type determines what values it can hold and what operations can be performed on it. Some valid variable names are: *sum*, *student\_age*, *totalMarks* etc. The rules for naming a variable are same as for the the identifiers which are already discussed in the previous



section. Blank space is not allowed in a variable name. Like other language C, C++ , the general syntax of the variable declaration in Java looks like this:

**type name ;**

---

### 2.4.1 THE SCOPE OF VARIABLES

---

In addition to the name and the type that we explicitly give to a variable, a variable has scope. The section of code where the variable's simple name can be used is the variable's scope. The variable's scope is determined implicitly by the location of the variable declaration, that is, where the declaration appears in relation to other code elements. We will learn more about the scope of Java variables in this section.

Java variables are categorized into three groups according to their scope. These are: **local variable** , **instance variable** and **class variable**.

- **Local variables** are variables which are declared and used inside methods. Outside the method definition they are not available for use and so they are called local variables. Local variables can also be declared inside program blocks that are defined between an opening { and a closing brace }. These variables are visible to the program only.
- **Instance variables** are created when the *class* objects are instantiated. They can take different values for each object.
- **Class variables** are global to a class and belong to the entire set of object that class creates. Only one memory location is reserved for each *class* variable. *Instance* and *class* variables are declared inside a class.

---

## 2.5 CONSTANTS

---

While a program is running, different values may be assigned to a variable at different times (thus the name *variable*, since the values it

contains can vary), but in some cases we don't want this to happen. If we want a value to remain fixed, then we use a constant. **Constants** in Java refer to fixed values that don't change during the execution of a program. A constant is declared in a manner similar to a variable but with additional reserved word **final**. A constant must be assigned a value at the time of its declaration. Thus, the general syntax is:

**final type name = value ;**

Here's an example of declaring some constants:

```
final double PI = 3.14159;
```

```
final int PASS_MARK = 200 ;
```

Java supports several types of constants such as *Integer constants*, *Real constants*, *Single Character constants*, *String constants*, *Backslash Character constants*.

- **Integer constant** : An integer constant refers to a sequence of digits. For example, Decimal integer constants: 5, -5, 0, 254321

Octal: 0, 025, 0125

Hexadecimal: 0X2, 0X9F, 0x etc.

- **Real constant** : Real or floating point constants are represented by numbers containing fractional parts. For example,

0.125, -.25, 124.75, .8, 450., -85 etc.

Exponential notation: 1.5e+4, 0.55e4 etc.

- **Single Character constant** : Character constant contains a single character enclosed within a pair of single quote marks. For example,

'1' , '15' , 'A', 'c' , ' ' , ' ; ' etc.

- **String constant** : It is a sequence of characters enclosed between double quotes. Characters may be alphabets, digits, special characters, blank spaces. For example,

"2010", "KKSHOU", "3+4", "Hello Java", "\$5", "A"

- **Backslash Character constant** : Java supports some backslash character constants that are used in output methods. These are also known as *escape sequences*. For example,

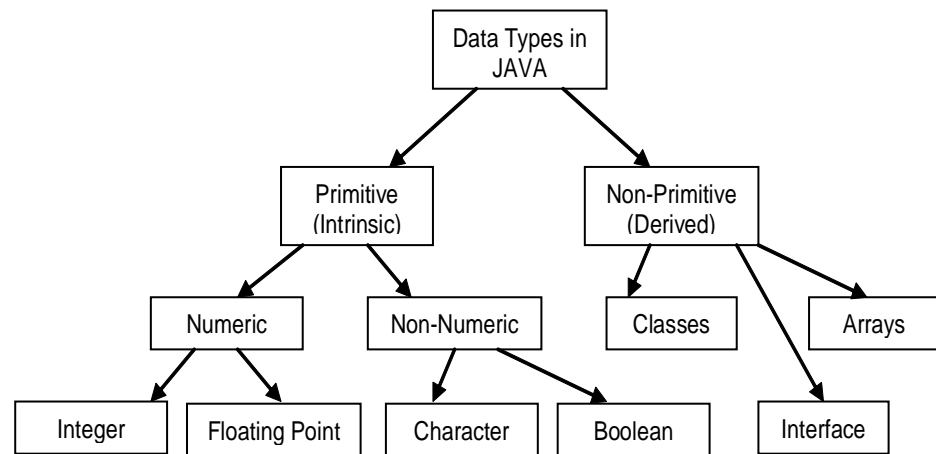
```
' \b '   back space
' \n '   new line
' \f '   form feed
' \r '   carriage return
' \t '   horizontal tab
' \' '   single quote
' \" \" ' double quote
' \\ '   backslash
```

---

## 2.6 DATA TYPES

---

Every variable must have a data type. A data type determines the values that the variable can contain and the operations that can be performed on it. A variable's type also determined how its value is stored in the computer's memory. The JAVA programming language has the following categories of data types(Fig 2.1):



**Fig. 2.1 : Data types in Java**

A variable of **primitive** type contains a single value of the appropriate size and format for its type: a number, a character, or a boolean value. Primitive types are also termed as *intrinsic* or *built-in* types. The primitive types are described below:

- **Integer type**

Integer type can hold whole numbers like 1,2, 3,... -4, 1996 etc. Java supports four types of integer types: **byte**, **short**, **int** and **long**. It does not support *unsigned* types and therefore all Java values are *signed* types. This means that they can be positive or negative.

For example, the **int** value 1996 is actually stored as the bit pattern 00000000000000000000000011111001100 as the binary equivalent of 1996 is 11111001100. Similarly, we must use a **byte** type variable for storing a number like 20 instead of an **int** type. It is because that smaller data types require less time for manipulation.

We can specify a long integer by putting an 'L' or 'l' after the number. 'L' is preferred, as it cannot be confused with the digit '1'.

- **Floating Point type**

Floating point type can hold numbers containing fractional parts such as 2.5, 5.75, -2.358. i.e., a series of digits with a decimal point is of type floating point. There are two kinds of floating point storage in Java. They are: **float** (Single-precision floating point) and **double** (Double-precision floating point).

In general, floating point numbers are treated as double-precision quantities. To force them to be in single-precision mode, we must append 'f' or 'F' to the numbers. For example, 5.23F, 2.25f

- **Character type**

Java provides a character data type to store character constants in memory. It is denoted by the keyword **char**. The size of char type is 2 bytes.

- **Boolean type**

Boolean type can take only two values: **true** or **false**. It is used when we want to test a particular condition during the execution of the program. It is denoted by the keyword **boolean** and it uses 1 byte of storage.

The memory size and range of all eight primitive data types are given in the following table 2.2 :

Type	Size	Minimum Value	Maximum 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
float	4 bytes	3.4e-038	3.4e+038
double	8 bytes	1.7e-308	1.7e+308
char	2 bytes	a single Unicode character	
boolean	1 byte	a boolean value (true or false)	

**Table 2.2: Size and Range of Primitive type**

In addition to eight primitive types, there are also three kinds of **non-primitive** types in JAVA. They are also termed as *reference* or *derived* types. The non-primitive types are: **arrays**, **classes** and **interfaces**. The value of a non-primitive type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable. These are discussed later as and when they are encountered.



### CHECK YOUR PROGRESS 1

- Which of the following are valid variable names?  
char, float, anInt, p, 4, total-marks, total\_matks, sum10digit, num2, MARKS, super  
.....  
.....  
.....
- What are the eight Java primitive types?  
.....  
.....  
.....

3. Write true or false :

- (i) A constant is declared in a manner similar to a variable but with additional reserved word *final*. (True /False)
- (ii) *Boolean* and it uses 2 bytes of storage. (True /False)
- (iii) Array is an example of non-primitive type. (True /False)
- (iv) Identifier name must not begin with a digit. (True /False)
- (v) Variable name may consists of only alphabets and digits. (True /False)

## 2.7 OPERATORS AND EXPRESSIONS

**Operators** are special symbols that are commonly used in expressions. An operator performs a function on one, two, or three operands. An operator that requires one operand is called a *unary operator*. For example, ++ is a unary operator that increments the value of its operand by 1. An operator that requires two operands is a *binary operator*. For example, = is a binary operator that assigns the value from its right-hand operand to its left-hand operand. And finally, a *ternary operator* is one that requires three operands. The Java programming language has one ternary operator (?:) .

**Expressions** are the simplest form of statement in Java that actually accomplishes something. *Expressions* are statements that return a value.

Many Java operators are similar to those in other programming languages. Java supports most C++ operators. In addition, it supports a few that are unique to it. Operators in Java include **arithmetic**, **assignment**, **increment** and **decrement**, **Relational** and **logical** operations. This section describes all these things.

**Arithmetic Operators** : Java has five operators for basic arithmetic (Table 2.3):

Operator	Meaning	Example
+	Addition	2 + 3
-	Subtraction	5 - 2
*	Multiplication	2 * 3
/	Division	14 / 4
%	Modulus	14 % 4

**Table 2.3. Arithmetic operators**

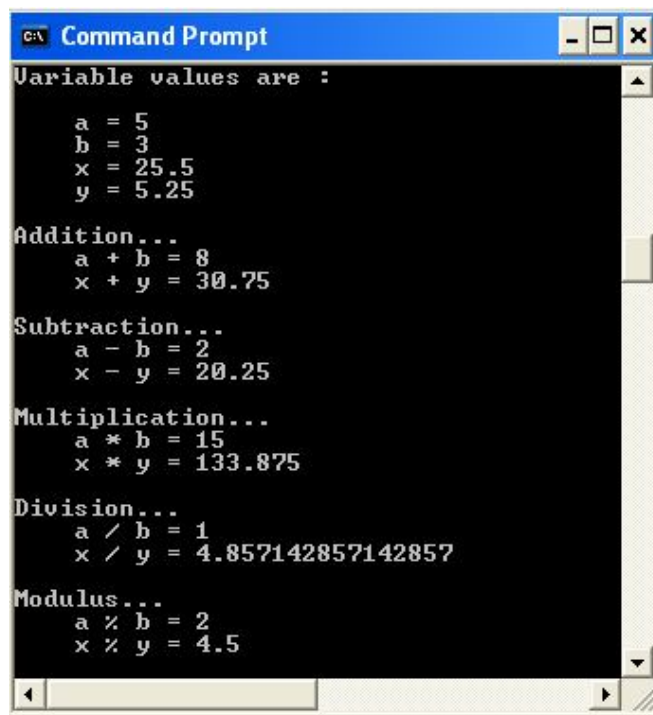
In the table, each operator takes two operands, one on either side of the operator. Integer division results in an integer. Because integers don't have decimal fractions, any remainder is ignored. The expression  $14 / 4$ , for example, results in 3. The remainder 2 is ignored in this case. Modulus (%) gives the remainder once the operands have been evenly divided. For example,  $14 \% 4$  results in 2 because 4 goes into 14 three times, with 2 left over. A sample program for arithmetic operation is given below:

**//Program 1: OperatorDemo.java**

```
public class OperatorDemo
{
    public static void main(String[ ] args){
        int a = 5, b =3;
        double x = 25.50, y = 5.25;
        System.out.println("Variable values are :\n");
        System.out.println("    a = " + a);
        System.out.println("    b = " + b);
        System.out.println("    x = " + x);
        System.out.println("    y = " + y);
        //Addition
        System.out.println("\nAddition...");
        System.out.println("    a + b = " + (a + b));
        System.out.println("    x + y = " + (x + y));
        //Subtraction
        System.out.println("\nSubtraction...");
        System.out.println("    a - b = " + (a - b));
        System.out.println("    x - y = " + (x - y));
        //Multiplication
        System.out.println("\nMultiplication...");
        System.out.println("    a * b = " + (a * b));
        System.out.println("    x * y = " + (x * y));
        //Division operation
        System.out.println("\nDivision...");
        System.out.println("    a / b = " + (a / b));
```

```
System.out.println("    x / y = " + (x / y));  
//Modulus operation  
System.out.println("\nModulus...");  
System.out.println("    a % b = " + (a % b));  
System.out.println("    x % y = " + (x % y));  
}  
}
```

The output of the above program will be:



```
Command Prompt  
Variable values are :  
    a = 5  
    b = 3  
    x = 25.5  
    y = 5.25  
  
Addition...  
    a + b = 8  
    x + y = 30.75  
  
Subtraction...  
    a - b = 2  
    x - y = 20.25  
  
Multiplication...  
    a * b = 15  
    x * y = 133.875  
  
Division...  
    a / b = 1  
    x / y = 4.857142857142857  
  
Modulus...  
    a % b = 2  
    x % y = 4.5
```

### Assignment Operators :

Assignment operators are used to assign the value of an expression to a variable. The usual assignment operator is '='. The general syntax is:

**variableName = value;**

For example, `sum = 0;` // 0 is assigned to the variable sum

`x = x + 1;`

Like C/C++, Java also supports the shorthand form of assignments. For example, the statement `x = x + 1;` can be written as `x += 1;` in shorthand form.



### **Increment and Decrement Operators :**

The unary **increment** and **decrement** operators ++ and -- comes in two forms, *prefix* and *postfix*. They perform two *operations*. They *increment* (or *decrement*) their operand, and *return* a value for use in some larger expression.

In **prefix** form, they modify their operand and then produce the new value. In **postfix** form, they produce their operand's original value, but modify the operand in the background. For example, let us take the following two expressions:

```
y = x++;
y = ++x;
```

These two expressions give very different results because of the difference between prefix and postfix. When we use postfix operators (x++ or x--), y gets the value of x before x is incremented; using prefix, the value of x is assigned to y after the increment has occurred.

### **Relational Operators :**

Java has several expressions for testing equality and magnitude. All of these expressions return a boolean value (that is, true or false). Table 2.4 shows the relational operators:

Operator	Meaning	Example
==	Equal	x == 5
!=	Not equal	x != 10
<	Less than	x < 7
>	Greater than	x > 4
<=	Less than or equal to	y <= 8
>=	Greater than or equal to	z >= 15

**Table 2.4. Relational operators**

### **Logical Operators :**

Expressions that result in boolean values (for example, the Relational operators) can be combined by using logical operators that represent the logical combinations **AND**, **OR**, **XOR**, and logical **NOT**.

For **AND** operation, the && symbol is used. The expression will be

true only if both operands tests are also true; if either expression is false, the entire expression is false.

For **OR** expressions, the || symbol is used. OR expressions result in true if either or both of the operands is also true; if both operands are false, the expression is false.

In addition, there is the **XOR** operator ^, which returns true only if its operands are different (one true and one false, or vice versa) and false otherwise (even if both are true).

For **NOT**, the ! symbol with a single expression argument is used. The value of the NOT expression is the negation of the expression; if x is true, !x is false.

### **Bitwise Operators :**

Bitwise operators are used to perform operations on individual bits in integers. Table 2.5 summarizes the bitwise operators available in the JAVA programming language. When both operands are boolean, the bitwise AND operator (&) performs the same operation as logical AND (&&). However, & always evaluates both of its operands and returns true if both are true. Likewise, when the operands are boolean, the bitwise OR (|) performs the same operation as is similar to logical OR (||). The | operator always evaluates both of its operands and returns true if at least one of its operands is true. When their operands are numbers, & and | perform bitwise manipulations.

Operator	Meaning
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero fill right shift
~	Bitwise complement
<<=	Left shift assignment ( $x = x \ll y$ )
>>=	Right shift assignment ( $x = x \gg y$ )
>>>=	Zero fill right shift assignment ( $x = x \ggg y$ )
$x \& y$	AND assignment ( $x = x \& y$ )
$x   y$	OR assignment ( $x = x   y$ )
$x \wedge y$	NOT assignment ( $x = x \wedge y$ )

**Table 2.5: Bitwise operators in Java**

A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left. For example if op1 and op2 are two operands, then the statement

```
op1 << op2;
```

shift bits of op1 left by distance op2; fills with zero bits on the right-hand side and  $op1 \gg op2$ ; shift bits of op1 right by distance op2; fills with highest (sign) bit on the left-hand side.

```
op1 >>> op2;
```

shift bits of op1 right by distance op2; fills with zero bits on the left-hand side. Each operator shifts the bits of the left-hand operand over by the number of positions indicated by the right-hand operand. The shift occurs in the direction indicated by the operator itself.

For example, the statement  $25 \gg 1$ ; shifts the bits of the integer 25 to the right by one position. The binary representation of the number 25 is 11001. The result of the shift operation of 11001 shifted to the right by one position is 1100, or 12 in decimal.

The Java programming language also supports the following operators (Table 2.6):

Operator	Description
?:	Conditional operator (a ternary operator)
[ ]	Used to declare arrays, to create arrays, and to access array elements
.	Used to form qualified names
(params )	Delimits a comma-separated list of parameters
(type )	Casts (converts) a value to the specified type
new	Creates a new object or array
instanceof	Determines whether its first operand is an instance of its second operand

Table 2.6: Other Operators



## CHECK YOUR PROGRESS 2

1. Write true or false :

- (i) Logical XOR operator returns true only if its operands are different.
- (ii) Logical AND operator returns true only if both operands tests are false.
- (iii) `x+=7;` has the same effect as `x = x+7;`
- (iv) `&&` is the symbol of logical AND and `&` is the symbol of bitwise AND operator.

2. Determine the value of each of the following logical expressions if `x = 5`, `y = 10` and `z = - 6`

- (i) `x > y && x < z`
- (ii) `x == z || y > x`
- (iii) `x < y && x > z`

## 2.8 CONTROL FLOW STATEMENTS

When we write a program, we type statements into a file. Without control flow statements, the compiler executes these statements in the order they appear in the file from left to right, top to bottom in a sequence. We can use control flow statements in our programs to

conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control.

Flow control in Java uses similar syntax as in C and C++. The Java programming language provides several control flow statements, which are listed below :

<b>Decision making</b>	: if, if-else, switch-case
<b>Looping</b>	: while, do-while, for
<b>Branching</b>	: break, continue, label :, return

---

### 2.8.1 Decision Making Statements

---

While programming, we have a number of situations where we may have to change the order of execution of statements based on certain conditions/decisions. This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly. The statements used to handle those situation are called decision making statement.

#### The *if* and *if-else* Statements

The *if* statement enables our program to selectively execute other statements, based on some criteria. The syntax of if statement is:

```
if (boolean_expression)
{
    statement-block ;
}
statement;
```

The *statement-block* may be a single statement or a group of statements. If the *boolean-expression* evaluates to true, then the block of code inside the *if* statement will be executed. If not the

first set of code after the end of the *if* statement(after the closing curly brace) will be executed. For example,

```
if (percentage>=40)
{
    System.out.println("Pass");
}
```

In this case, if **percentage** contains a value that is greater than or equal to 40, the expression is true, and **println( )** will execute. If **percentage** contains a value less than 40, then the **println( )** method is bypassed. What if we want to perform a different set of statements if the expression is false? We use the *else* statement for that.

The general syntax of ***if-else*** statement is:

<pre>if ( Boolean_expression )     statement; //executes when the expression is true else     statement; //executes when the expression is false</pre>
--

Let us consider the same example but at this time the output should be *Pass* or *Fail* depending on percentage of marks. i.e., if percentage is equal to or more than 40 then the output should be *Pass*; otherwise *Fail*. This can be done by using an *if* statement along with an *else* statement. Here is the segment of code :

```
if (percentage>=40)
    System.out.println("Pass");
else
    System.out.println("Fail");
```

When a series of decisions are involved, we may have to use more than one *if-else* statements in nested form.

### **The switch statement**

We have seen that when one of the many alternatives is to be selected, we can design a program using *if* statements to control the selection. However, the program becomes difficult to read and follow when the number of alternatives increases. Like C/C++, JAVA has a built-in multiway decision statement known as a ***switch***.

The *switch* statement provides variable entry points to a block. It tests the value of a given *variable* or *expression* against a list of **case** values and when a match is found, a block of statements associated with that *case* is executed. The general form of *switch* statement is as follows :

```
switch( expression )
{
    case value : statements;
                    break;
    case value : statements
                    break;
    ...
    default : statements // optional default section
                    break;
}
```

The *expression* is evaluated and compared in turn with each *value* prefaced by the *case* keyword. The values must be *constants* (i.e., determinable at compile-time) and may be of type byte, char, short, int, or long.

---

### 2.8.2 Looping

---

In looping, a sequence of statements are executed until some conditions for the termination of the loop are satisfied. The process of repeatedly executing a block of statements is known as *looping*. At this point, we should remember that Java does not support **goto** statement. Like C/C++, Java also provides the three different statements for looping. These are:

- **while**
- **do-while**
- **for**

#### The *while* and *do-while* statements

We use a *while* statement to continually execute a block while

a condition remains true. The general syntax of the *while* statement is:

```
while(expression)
{
    statement(s);
}
```

First, the *while* statement evaluates *expression* , which must return a boolean value. If the expression returns *true*, the *while* statement executes the statement(s) in the while block. The *while* statement continues testing the expression and executing its block until the expression returns *false*.

The Java programming language provides another statement that is similar to the *while* statement : the ***do-while*** statement. The general syntax of *do-while* is:

```
do
{
    statement(s);
}while (expression);
```

Statements within the block associated with a *do-while* are executed at least once. Instead of evaluating the expression at the top of the loop, *do-while* evaluates the expression at the bottom. Here is the previous program rewritten to use *do-while* loop.

A program of *while* loop is shown below :



**Program 2.2: Fibo.java**

```
class Fibo

{
    public static void main(String args[])
    {
        System.out.println("0\n1");
        int n0=0,n1=1,n2=1;
        while(n2<50)
        {
            System.out.println(n2);
            n0=n1;
            n1=n2;
            n2=n1+n0;
        }
        System.out.println(n2);
    }
}
```

The output of the above program will be the Fibonacci series between 0 to 50 (i.e., 0 1 1 2 3 5 8 13 21 34).

**The for statement**

The *for* statement provides a compact way to iterate over a range of values. The general form of the *for* statement can be expressed like this:

<pre><b>for (initialization; termination_condition; increment)</b> {     <b>statement(s);</b> }</pre>
---

The *initialization* is an expression that initializes the loop. It is executed once at the beginning of the loop. The *termination\_condition* determines when to terminate the loop. This condition is evaluated at the top of each iteration of the

loop. When the condition evaluates to *false*, the loop terminates. Finally, *increment* is an expression that gets invoked after each iteration through the loop. All these components are optional. In fact, to write an infinite loop, we can omit all three expressions:

```
for ( ; ; )
```

```
{  
    // infinite loop  
}
```

Often, *for* loops are used to iterate over the elements in an array or the characters in a string. The following program segment uses a *for* loop to calculate the summation of 1 to 50:

```
for (i =1; i <= 50 ; i + +)  
{  
    sum = sum + i ;  
}
```

---

### 2.8.3 BRANCHING STATEMENTS

---

The Java programming language supports three branching statements:

- The ***break*** statement
- The ***continue*** statement
- The ***return*** statement

#### The ***break*** statement

In JAVA, the *break* statements has two forms: *unlabelled* and *labelled*. We have seen the unlabelled form of the *break* statement used with *switch* earlier. As noted there, an unlabelled *break* terminates the enclosing *switch* statement, and the flow of control transfers to the statement immediately following the *switch*. It can be used to terminate a *for*, *while*, or *do-while* loop. A ***break*** (unlabelled form) statement, causes an immediate jump out of a loop to the first statement after its end. When the *break* statement is encountered inside a loop, the loop is immediately

exited and the program continues with the statement immediately following the loop. When the loop is nested, the *break* would only exit from the loop containing it. This means, the *break* will exit only a single loop.

### The *continue* statement

The ***continue*** statement causes an immediate branch to the end of the innermost loop that encloses it, skipping over any intervening statements. It is written as:

**continue ;**

A *continue* does not cause an exit from the loop. Instead, it immediately initiates the next iteration. We can use the *continue* statement to skip the current iteration of a *for*, *while*, or *do-while* loop.

In Java, we can give a label to a block of statements. A label is any valid Java variable name. To give a label to a loop, we have to place the label name before the loop with a colon at the end. For example,

```
loop1: for( ..... )
{
    .....
    .....
}
.....
```

We have seen that a simple *break* statement causes the control to jump outside the nearest loop and a simple *continue* statement returns the current loop. If we want to jump outside a nested loop or to continue a loop that is outside the current one, then we may have to use the *labelled break* and *labelled continue* statement. The labelled form of *break* and *continue* can be used as follows:

**Program 2.3:** breakDemo.java

```
class breakDemo
{
    public static void main(String[] args)
    {
        outer: for(int i=1;i<100;i++)
        {
            System.out.println(" ");
            if(i>=10)
                break;
            for( int j=1;j<100;j++)
            {
                System.out.print("* ");
                if(j==i)
                    continue outer; //labelled continue
            }
        }
    }
}
```

The output of the above program will be like this:

**The return statement**

The last of the branching statements is the *return* statement. We can use *return* to exit from the current method. The flow of control returns to the statement that follows the original method call. The *return* statement has two forms: one that returns a value and one that does not.

To return a value, simply put the value (or an expression that calculates the value) after the return keyword:

```
return sum;
```

The data type of the value returned by return must match the type of the method's declared return value. When a method is declared void, use the form of *return* that doesnot return a value:

```
return;
```



### CHECK YOUR PROGRESS 3

1. What is wrong with this code :

```
switch(n){  
  
    case 1: a=5;  
           b=10;  
           break;  
  
    case 2:  
           c=15;  
           break;  
           d=20;  
  
}
```

2. Explain the difference between these three blocks of code

(i) if (a>5)

```
    if(a<10) System.out.println(a);
```

(ii) if (a>5) System.out.println(a);

```
    if (a<10) System.out.println(a);
```

(iv) if (a>5) System.out.println(a);

```
    else System.out.println(a);
```

3. Write true or false:

(i) A program stops its execution when a break statement is encountered.

(ii) One *if* can have more than one *else if* clause.

(iii) A *continue* cause an exit from the loop

(iv) A *break* statement causes an immediate jump out of a loop to the first statement after its end.

4. Write a Java program to display the multiplication table of a particular number using for loop.

.....  
 .....  
 .....  
 .....

## 2.9 LET US SUM UP

In this unit we have discussed all the basic data types and operators in Java and their use in expressions. The control flow statements are the backbone of any programming language. Here, we have covered the discussion of *if*, *if-else*, *while*, *do-while* and *for* statements with their appropriate syntax. We have also seen how to use the *break* and *continue* statements to skip or jump out of loop, if need be. We have learnt to use the labelled form of *break* and *continue* in Java programming.

The key points you are to keep in mind in this unit are:

- Java variables are categorized into three groups according to their scope: *local variable* , *instance variable* and *class variable*.
- In Java, constants are declared in the manner similar to variables but with additional reserved word *final*.
- *Goto* statement is not supported by Java programming language.
- In Java, *break* and *continue* statements can be used with a *label*.