KRISHNA KANTA HANDIQUE STATE OPEN UNIVERSITY

Housefed Complex, Dispur, Guwahati - 781 006

MASTER OF COMPUTER APPLICATION

PROGRAMMING IN JAVA

CONTENTS

Subject Experts

Prof. Anjana Kakati Mahanta, Deptt. of Computer Science, Gauhati University
Prof. Jatindra Kr. Deka, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati
Prof. Diganta Goswami, Deptt. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati

Course Co-ordinator

Tapashi Kashyap Das, Assistant Professor, Computer Science, KKHSOU
Arabinda Saikia, Assistant Professor, Computer Science, KKHSOU

SLM Preparation Team

Units	Contributors
1, 4, 5	**Arabinda Saika,** KKHSOU
2, 3,6	**Tapashi Kashyap Das,** KKHSOU
7, 8, 9, 10	**Prnajit Baruah** **Technical Manager** **Zaloni International Pvt. Ltd, Guwahati**

January 2013

Printed and published by Registrar on behalf of the Krishna Kanta Handique State Open University.

The University acknowledges with thanks the financial support provided by the **Distance Education Council, New Delhi**, for the preparation of this study material.

Housefed Complex, Dispur, Guwahati - 781006; Website : www.kkhsou.in

COURSE INTRODUCTION

This is a course on "***Programming in Java***". This course is designed as an introduction to Java Programming. Java was developed at Sun Microsystems. Work on Java originally began with the goal of creating a platform-independent language and operating system for consumer electronics. What we know today as Java is both a programming language and an environment for executing programs written in the Java language. Java has become very popular because of its natural applicability to programming network software for the World Wide Web (WWW). Java is best known as a language for programming *applets*. An applet is a "miniapplication" that runs within the context of a larger application, such as a network browser, which facilitates the supporting of animation, graphics, games and wide range of special effects.

This block comprises the following five units :

Unit - 1 introduces the basics of Java programming. This unit also discusses how to install Java Development Kit into your computer.
Unit - 2 concentrates on the discussion of *Java tokens, keywords, operators* and *variables*. In addition, this unit elaborates on the decision making and control statements.
Unit - 3 describes the object oriented features of Java. The concept of *class, object, methods*, *constructors and modifiers* are elaborated in this unit. Moreover, an important propery of an object oriented language, *inheritance,* is illustrated here.
Unit - 4 discusses the three important concept of Java - Arrays, Strings and Vectors.
Unit - 5 is the last unit of this block. This unit focuses on the creation and use of the Java interfaces and Java packages.
Unit - 6 discusses how to handle exceptions in Java. Examples of different built-in exceptions are presented in this unit. At the end, how can we use user-defined exception is discussed.
Unit - 7 deals with one most important Java package .io and the concept of file handling. How to read console input and how to write console output are also covered in this unit.
Unit – 8 introduces the concept of *Applets* which can be included in an HTML page. How programs can be written for Internet application is discussed in this unit.
Unit - 9 explains the AWT and Swing which are two important paint mechanism in Java. The AWT supports GUI Java programming. Its purpose is to help developers write correct and efficient GUI painting code.
Unit - 10 is the last unit of this block. This unit introduces you with JDBC which is a programming interface between Java programs and DBMS.

Each unit of these blocks includes some along-side boxes to help you know some of the difficult, unseen terms. Some "EXERCISES" have been included to help you apply your own thoughts. You may find some boxes marked with: "LET US KNOW". These boxes will provide you with some additional interesting and relevant information. Again, you will get "CHECK YOUR PROGRESS" questions. These have been designed for a self-check of your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with "ANSWERS TO CHECK YOUR PROGRESS" given at the end of each unit.

MASTER OF COMPUTER APPLICATION

PROGRAMMING IN JAVA

DETAILED SYLLABUS

Unit - 1 **Introduction to JAVA Marks : 8**

An overview of JAVA, Basic features of Java, JAVA Environment, Installing the Java SDK, Writing Java Programs.

Unit - 2 **Programming Basic Marks : 12**

Java Token & Keywords, Constants, Data types; Declaring a variable, The scope and lifetime of variable, Various Operators, Decision Making and Control Statements : if statement, If-else, else-if, switch statement; the for, while, do-while statements.

Unit - 3 **OOP in Java Marks : 15**

Class fundamentals : Defining class, Accessing class members, Declaring objects, Constructors, copy constructor; Passing Arguments to Methods, Modifiers, Inheritance : the super class, Multilevel Inheritance, Final and abstract keyword, Static Members.

Unit - 4 **Arrays, Strings and Vectors Marks : 10**

Declaring Arrays, Creating Arrays, Initializing Arrays, Multi-Dimensional Arrays, Strings: string arrays, string methods, StringBuffer class, Vectors.

Unit - 5 **Interfaces and Packages Marks : 8 P**

Interfaces: Defining an Interface, Implementing interfaces, Applying Interfaces, Packages: Defining a package, Accessing and Importing Packages.

Unit - 6 **Exception Handling (Marks : 9)**

Exception Handling fundamentals, Exception types, Using *try* and *catch,* built-in exceptions in Java, User-defined exception

Unit - 7 **File Handling (Marks : 8)**

I/O Basics: Streams, The Stream classes, The predefined streams, Reading console input, Writing console output, Reading and writing files

Unit – 8 **Introduction to Applets (Marks : 12)**

Applets and the World Wide Web, The Applet Class, Applets and HTML , The Life Cycle of an Applet, Using Window Components, Event Handling, Adding Audio and Animation

Unit – 9 **AWT and Swings (Marks : 10)**

AWT Basics, AWT Components, Event Handling, Application and Menus; Introduc tion to Swings, Swing Components, Event Handling, Display text and image in a window, Layout manager.

UNIT – 10 **Introduction to JDBC (Marks : 8)**

Basic steps to JDBC, API, JDBC Drivers, Connection Management, JDBC Design Considerations, Two Tier and Three Tier client server model, Resultset, Prepared statement and callable statement, Resultset MetaData Object

UNIT 1: INTRODUCTION TO JAVA

UNIT STRUCTURE

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.

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.

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.

- **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.

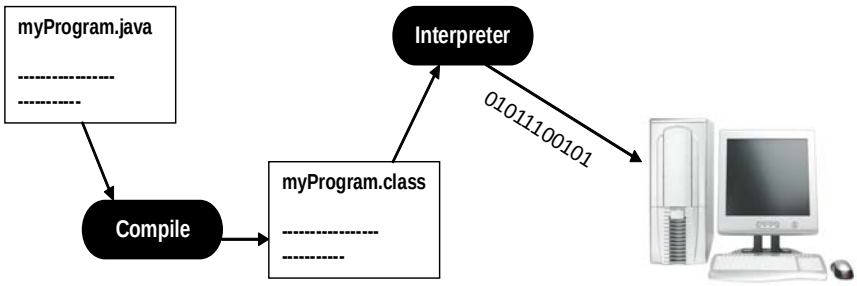


Fig. 1.1 A Java program first compiled and then interpreted

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.

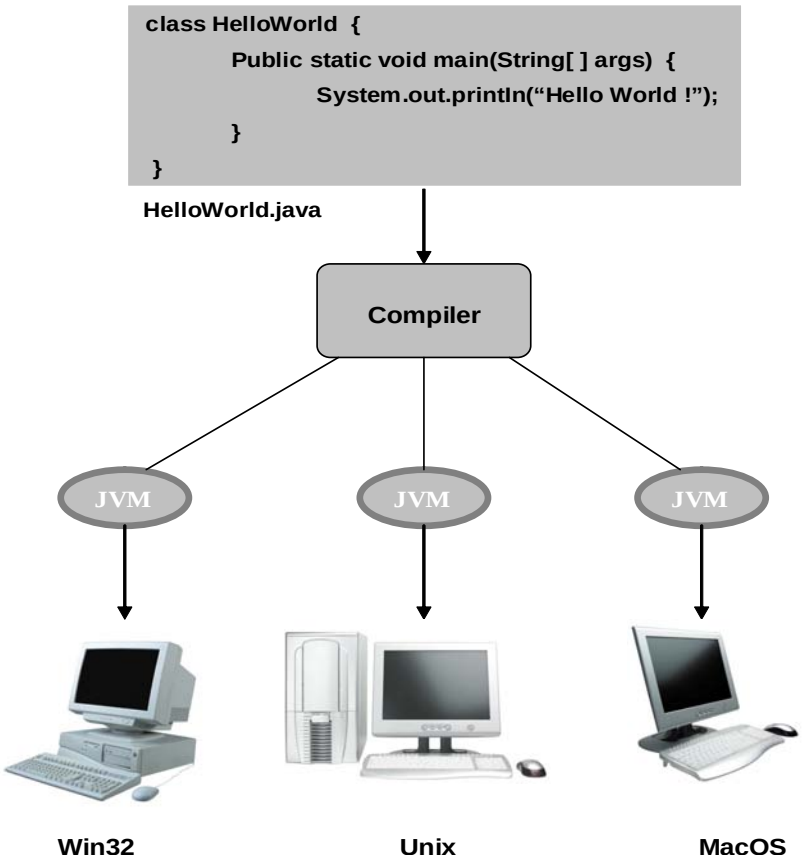


Fig1.2 Program written once and can run on almost any plateform

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 Inteface* (API).

Java development Kit

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

java The loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler.

javac The compiler, which converts source code into Java bytecode

jar The archiver, which packages related class libraries into a single JAR file.

javadoc The documentation generator, which automatically generates documentation from source code comments

jdb The Java debugger

jps The process status tool, which displays process information for current Java processes

javap The class file disassembler

appletviewer This tool can be used to run and debug Java applets without a web browser.

javah 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.

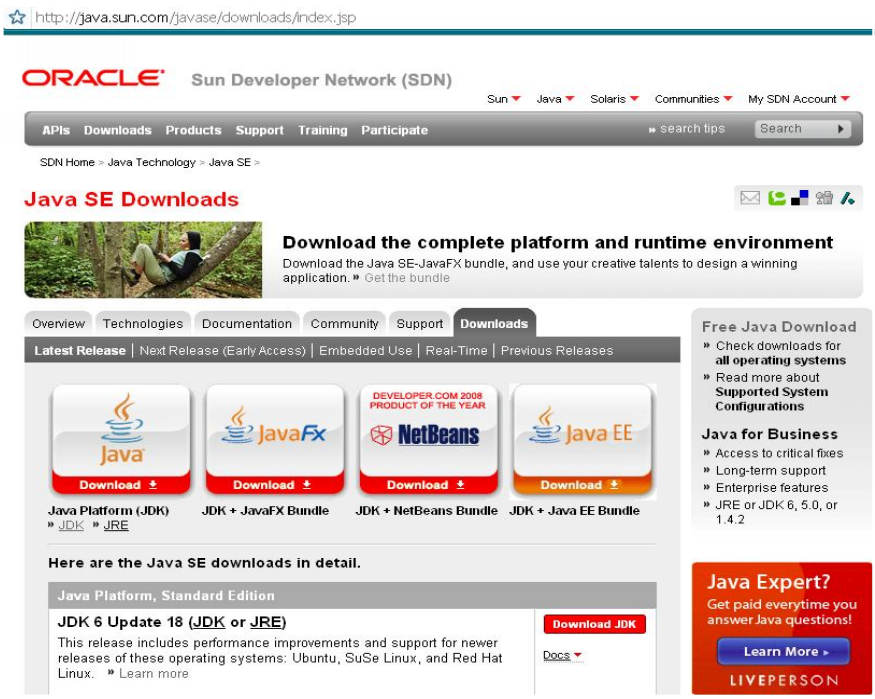


Fig 1.3 Download page for JDK

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 diectory 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.

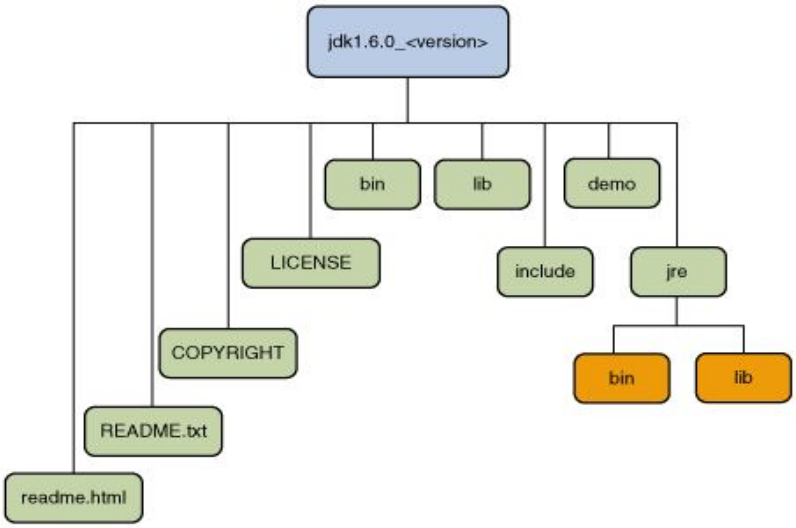


Fig. 1.4 JDK directory structure

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:> "\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

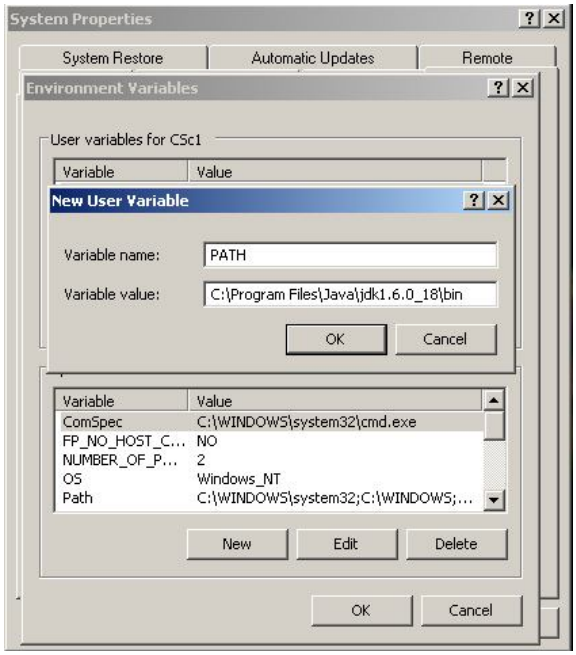


Fig 1.5 Setting the User variable

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 :

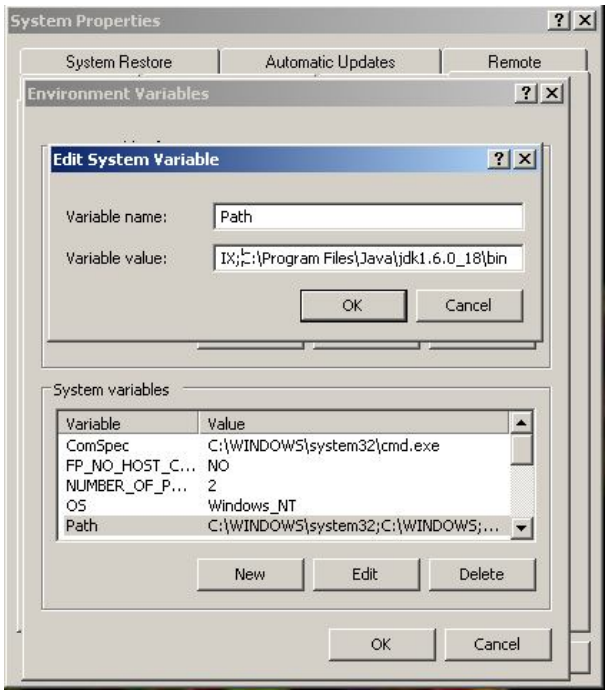


Fig. 1.6 Setting the System Variable

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.

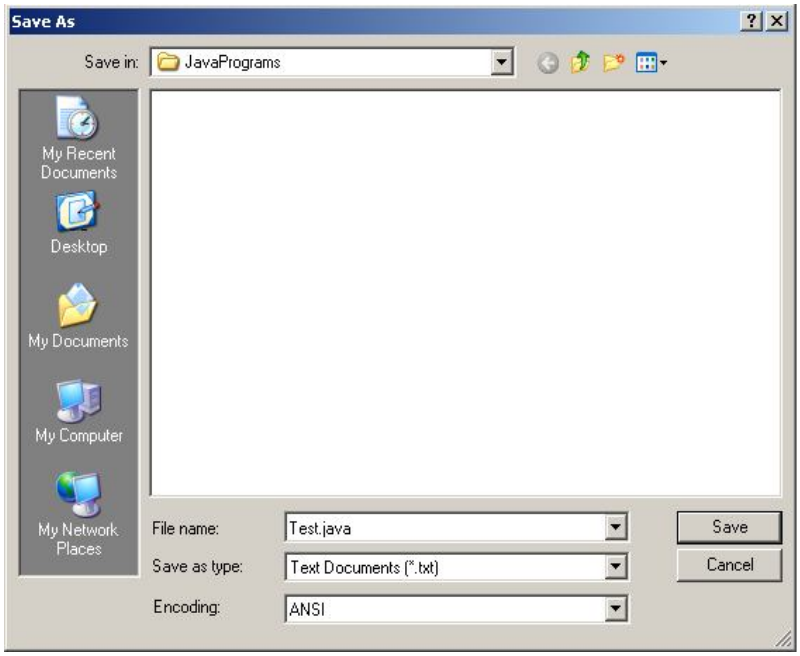


Fig.1.7 Saving Test.java file

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 looks 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.

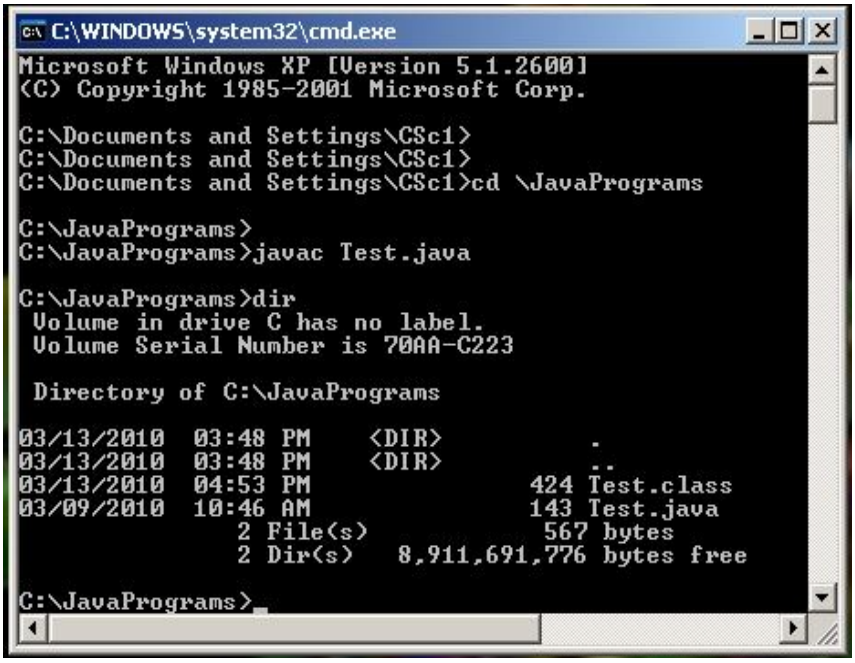


Fig. 1.8 Compiling a Java Program

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

java Test

Fig. 1.9 shows the result what you will see -

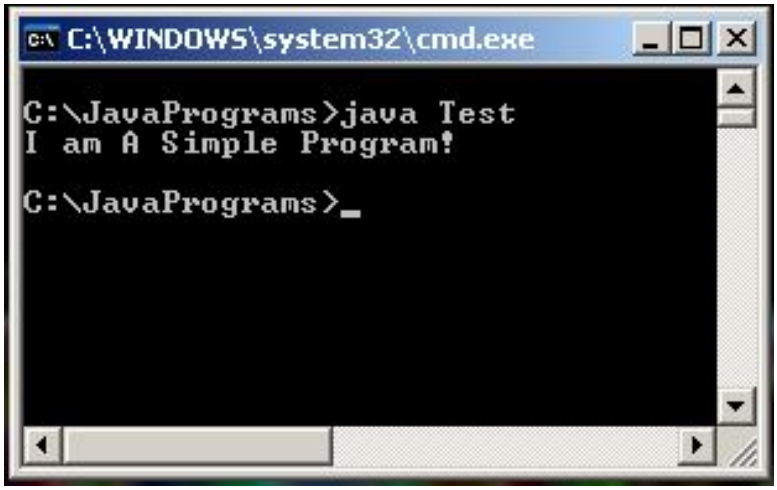


Fig. 1.9 Running a Java Program

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 ?

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

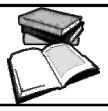
1.10 LET US SUM UP

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.

2. Why Java is called a platform independent language ?
3. How Java is mmore secure than other programming language ?
4. What is a Java applet ? What is its function ?
5. How is Java strongly associated with the Internet ?
6. What is the task of the main() function in a Java program ?
7. Describe the typical structure of a Java program.
8. Write short notes onthe following :
    a) Java virtual machine
    b) Java applets
9. Write and run a program that initializes a ***String*** object with the text 'KKHSOU' and then print it on three separate line.
10. Write and run a Java program that prompt the user for his or her last name and first name separately and then prints the name like this :

```
Enter your last name : Barman
Enter your first name : Hemanga

Hello, Hemanga Barman
```

*****

UNIT 2 PROGRAMMING BASIC

UNIT STRUCTURE

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 installantion 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 thse word 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	syncrhronized	this
threadsasafe	throw	throws	transient	true
try	var	void	volatile	while

Table 2.1 : Java Keywords

- **Identifiers :** Java Identifers 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. Underscore 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 opeates 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 donot 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 donot 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 contants such as *Integer constants*, *Real constants*, *Single Character constants*, *String constants*, *Backslash Character constants*.

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

  Octal: 0, 025, 0125

  Hexadecimal: 0X2, 0X9F, Ox 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):

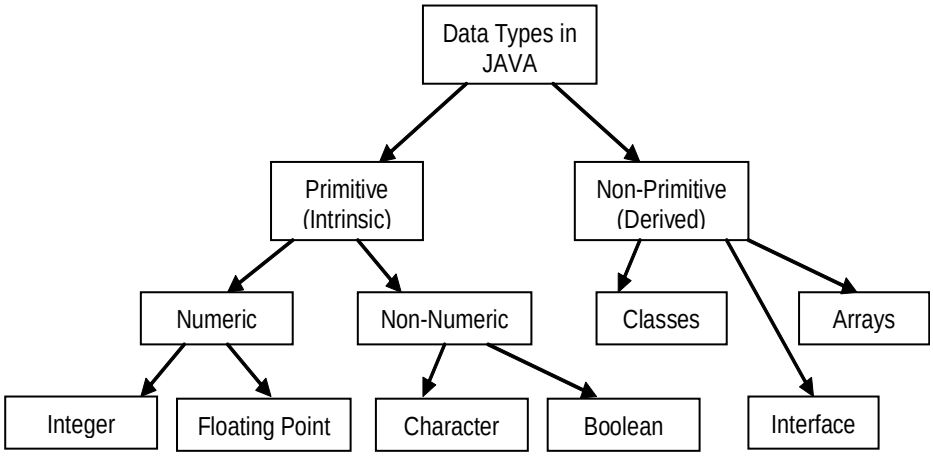


Fig. 2.1 : Data types in Java

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 00000000000000000000011111001100 as the binary equivalent of 1996 is 11111001100. Similary, 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 contaning fractorial 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

1. Which of the following are valid variable names?
   char, float, anInt, p, 4, total-marks, total_matks, sum10digit, num2, MARKS, super

   .....

   .....

   .....

2. 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

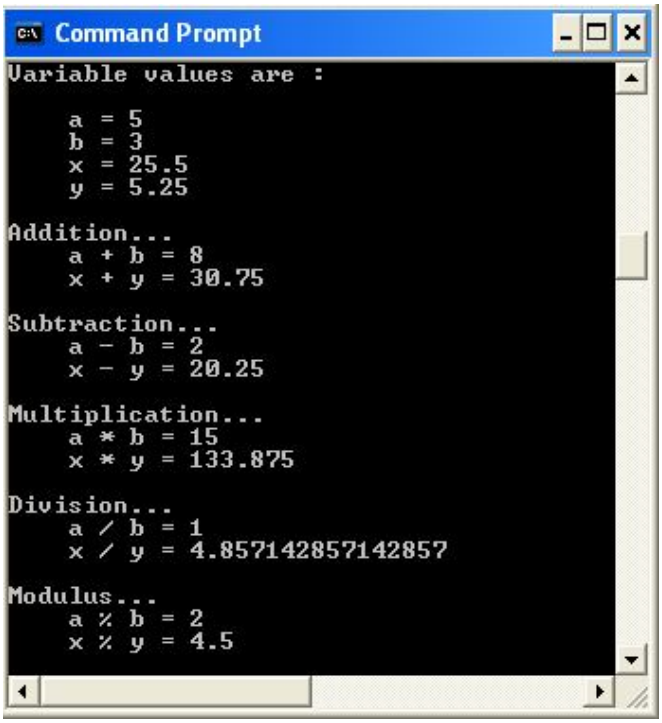
In the table, each operator takes two operands, one on either side of the operator. Integer division results in an integer. Because integers don not have decimal fractions, any remainder is ignored. The expression 14 / 4, for example, results in 3. The remainder 2 is ignored in this case. Modulus (%) 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:



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 aslo 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 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 << y)
>>=	Right shift assignment (x = x >> y)
>>>=	Zero fill right shift assignment (x = x >>> y)
x&=y	AND assignment (x = x & y)
x|=y	OR assignment (x + x | y)
x^=y	NOT assignment (x = x ^ 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 >> 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 >> 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 :

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

2.8.1 Decision Making Statements

While programing, we have a number of situations where we may have to change the order of execution of statements based on certain cosditions/decisions. This involves a kind of decision making to see whether a particular condition has occured 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:

```
if ( Boolean_expression )
   statement; //executes when the expression is true
else
   statement; //executes when the expression is false
```

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:

```
for (initialization; termination_condition; increment)
{
     statement(s);
}
```

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 doesnot.

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);
(i) 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 yoy 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.*

2.10 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress- 1

1. anInt, p, total_marks, sum10digit, num2, MARKS
2. The eight Java primitive types are boolean, char, byte, short, int, long, float and double
3. (i)True, (ii) False (iii) True (iv) True (v)False

Check Your Progress- 2

1. (i) True (ii) False (ii)True (iv) True

2.(i) FALSE (ii) TRUE (iii) TRUE

Check Your Progress - 3

1. The statement d=20; is unreachable
2. (i) the value of a is printed if a is greater than 5 but less than 10 (i.e., 5 < a <10).
   (ii) the value a is printed if a>5, and then again if a<10.
   (iii) the value a is printed regardless of its value

3. (i) False (ii) True (iii) False (iv) True

4.
```
class Multi
 {
    public static void main(String args[])
    {
        System.out.println("\nMultiplication Table of 5\n");
   int i,n=5;
          for(i=1;i<=10;i++)
           {
       System.out.println(n+"*"+i+"="+n*i);
           }
     }
  }
```

2.11 FURTHER READINGS

1. "*The Complete Reference -Java 2* " by Herb Hchildt, McGraw-Hill
2. "*JAVA How to Program*", Deitel & Deital, PHI Publication
3. "*Programming with JAVA- a Primer* ", E.Balagurusamy, TATA McGRAW Hill Publication.

2.12 MODEL QUESTIONS

1. (a) Consider the following code :

```
int x = 10;
int n = x++%5;
```

What are the values of x and n after the code is executed ?What are the final values of x and n if instead of using the postfix increment operator (x++), you use the prefix version (++x)?

(b) What is the value of i after the following code executes?

```
int i = 8;
i >>=2;
```

(c) What is the value of i after the following code executes?

```
int i = 17;
i >>=1;
```

2. What are the rules for naming a Java varibles?

3. What do you mean by looping? What are the different types of loop in the Java programming language. Briefly explain with their syntax.

4. What are the branching statements associted with the language Java?

5. Compare the statements *break* and *continue* in terms of their functions.

6. What are the different types of constants in Java? Explain with examples.

7. What is scope of a variable?
8. Write a program that computes and displays the factorial for any given number n.
9. Write a program to compute the sum of the digits of a given integer number.
10. Write a program using while loop to reverse the digits of a given number. For example, the number 2345 should be written as 5432.
11. Write a Java program to print the following output using for loop:

    1
    2 2
    3 3 3
    4 4 4 4
    5 5 5 5 5

12. Write a Java program that will reverse a given number declared as a constant in the program and display the reversed number.

*****

UNIT 3: OOP IN JAVA

UNIT STRUCTURE

3.1 LEARNING OBJECTIVES

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

- learn how to write your own classes in Java
- access class members in Java
- learn to declare objects, pass arguments to methods
- learn to use recursive methods in Java
- learn the different types of Java modifiers
- describe and use constructors, copy constructor etc.
- learn and use the concept of inheritance in Java
- learn about abstract method and abstract class
- describe static members

3.2 INTRODUCTION

We have already learnt the concepts of tokens, data types, variables, constants, decision and control statements, operators etc. along with their types as they relate to Java language.

Anything we wish to represent in a Java program must be encapsulated in a class. Classes are the building blocks of a Java application. In this unit, we will explore the object-oriented aspects of Java. We will learn about the concept of classes in Java, how to access class members, how to create instance of classses etc. Some other important concepts like inheritance, super classes, contructors etc. are also covered in this unit. At the end of this unit, we will acquiant you with the concept of recursion.

3.3 CLASS FUNDAMENTALS

Classes provide a convenient method for grouping together logically related data items and functions that work on them. A class can contain methods or functions, variables, initialization code etc. In case of Java, the data items are termed as ***fields*** and the functions are termed as ***methods***.

Class serves as a blueprint for making class *instances*, which are runtime objects that implement the class structure. Thus, an ***object*** is defined to be an instance of a class. An object consists of ***attributes*** and ***behaviors***. An attribute is a feature of the object, something the object “has.” A behavior is something the object “does”.

3.3.1 DECLARING AND DEFINING A CLASS

A class in Java is declared using the ***class*** keyword. A source code file in Java can contain exactly one public class, and the name of the file must match the name of the public class with a ***. java*** extension.

We can declare more than one class in a . java file, but at most one of the classes can be declared *public*. The name of the

source code file must still match the name of the public class. If there are no public classes in the source code file, the name of the file is arbitrary. The general form of a **class** definition is as follows:

```
class classname
{
        type instanceVariable1;
        type instanceVariable2;
        ..............
        ..............
        type instanceVariableN;
        type methodname1(parameterList)
        {
                // body of method
        }
        type methodname2(parameterList)
        {
                // body of method
         }
         ..............
         ..............
          type methodnameN(parameterList)
        {
                // body of method
        }
}
```

Here, we can see that, there is no semicolon after closing brace of class in Java. But in case of C++ language, class definition ends with a semicolon. The data or variables, defined within a class are called ***fields*** or ***instance variables*** as each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. The ***methods*** are used

for manipulating the instance variables(data items) contained in the class. The methods and variables defined within a class are collectively called *members* of the class. A class may contain three kinds of members: *fields*, *methods* and *constructors*. Constructor specify how the objects are to be created.

Java classes do not need to have a ***main( )*** method. The general form of a class does not specify a ***main( )*** method. We only specify one if that class is the starting point for our program. The fields and methods of a class appear within the curly brackets of the class declaration.

A class may not contain anything inside the curly brace { and }. Thus we can have empty classes in Java. An empty class cannot do anything as it does not contain any properties. But it is possible to create objects of such classes. The structure of an empty class is as follows:

```
class classname
{
}
```

Let us consider another class *Triangle* which has two float type instance variables *base* and *height*.

```
class Triangle
{
        float base;     //instance variable base
          float height;     //instance variable height
}
```

Until object of *Triangle* class is created, there is no storage space has been created in the memory for these two variables *base* and *height*. When an object of *Triangle* class is created, memory will be allocated for each of these two fields.

3.3.2 ADDING METHODS TO A CLASS

A ***method*** is a sequence of declarations and executable statements encapsulated together like independent mini program. For the manipulation of data fields inside a class we have to add methods into the class. Method contains local variable declarations and other Java statements that are executed when the method is invoked.

- The name of the method, which can be any valid identifier
- Return value (i.e., the type of the value the method returns)
- List of parameters, which appears within parentheses
- Definition of the method (i.e., the body of the method)

A Java application must contain a main() method whose signature looks like this :

```
public static void main(String args[ ])
 {
      // body;
 }
```

The meaning of the keywords *public*, *static* etc. are already outlined in *Unit-1*. *void* indicates that the *main()* method has no return value.

In Java, the *definition* (often referred to as the *body*) of a method must appear within the curly brackets that follow the method declaration. For example, let us add two methods in the *Triangle* class.

```
class Triangle

{
      float base, height;
```

```
        void readData(float b, float h)    //definition of method
        {
                base=b;
                height=h;
         }
       float findArea( )//definition of another method
        {
                float area = 0.5*(base *height);
                return area;
        }
}
```

In the above code, the *Triangle* class demonstrates how to add methods to a class. In the above class there are two methods, ***readData( )*** and ***findArea( )***. The method ***readData()*** has a return type of ***void*** because it does not return any value. We pass two float type values to the method which are then assigned to the instance variables *base* and *height*. The method ***findArea()*** calculates area of the triangle and returns the result. Here, we can directly use ***base*** and ***height*** inside the method *readData()* and *findArea()*.

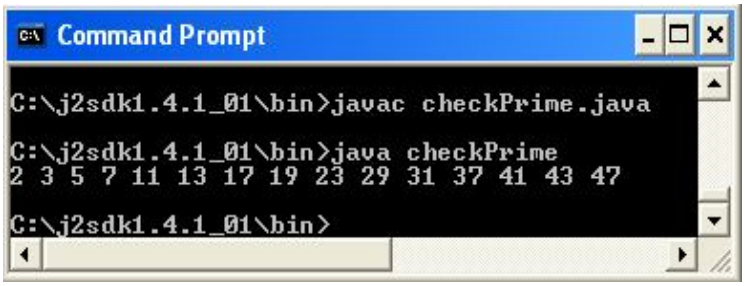
Let us consider another method ***primeCheck()*** for the illustration of method. The following program tests a boolean method that checks its arguments for primality. The main() method prints those integers for which the checkPrime() returns true:

Program 3.1: checkPrime.java

```
class  checkPrime
{
    public static void main(String [ ] args)
    {
          for(int i=0;i<50;i++)
                if(isPrime(i))
```

```
                    System.out.print(i+ " ");
            System.out.println();
    }
    static boolean isPrime(int n)   //static method
    {
            if(n<2)
                return false;
            if(n==2)
                return true;
            if(n%2==0)
                return false;
            for(int j=3;j<=Math.sqrt(n);j=j+2)
                if(n%j==0)
                    return false;
            return true;
    }
}
```

The output will give the prime numbers from 2 to 47 as follows:



3.3.3 CREATING OBJECTS

It is important to remember that a class declaration only creates a template; it does not create an actual object. Java allocates storage for an object when we create it with the ***new*** operator. Creating an object is also referred to as instanting an object. The new operator creates an object of the specified class and returns a reference to that object.

Thus, the preceding code does not cause any objects of type ***Triangle*** to come into existence. To create a **Triangle** object, we will use a statement like the following:

Triangle t1 = new Triangle( ); //t1 is an object of Triangle class

Here, the *Triangle( )* is the default constructor of the class. The above statement can also be written by splitting it into to two statements. This can be written as follows :

Triangle t1; // declares variable t1 to hold the object reference

t1 = new Triangle(); //assigns the object reference to the //variable t1

Again, each time we create an instance of a class, we are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Triangle** object will contain its own copies of the instance variables **base** and **height**.

3.3.4 ACCESSING CLASS MEMBERS

Now we can say that when an object is created, each object contains its own set of variables. After creating objects of a class, we should assign values to these variables in order to use them in our program. Instance variables and methods cannot be accessible directly outside the class. For this, we have to use concerned object and the *dot(.)* operator. The dot operator links the name of the object with the name of an instance variable or method. The syntax of accessing class member is as follows:

objectName.instanceVariableName ;
objectName.methodName(parameterList);

For example, suppose we have created **t1** object with the following statement

Triangle t1 = new Triangle( );

Now to assign the value 20.5 to the **base** variable of **t1**, we would use the following statement:

t1.base = 20.5;

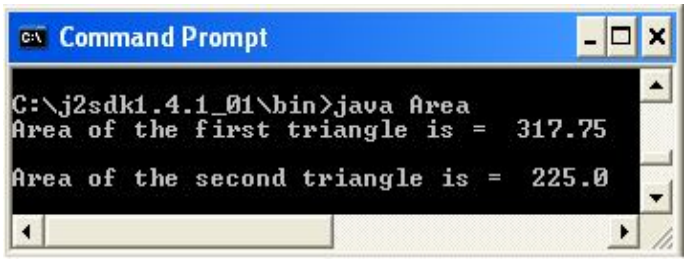
This statement tells the compiler to assign the copy of *base* that is contained within the t1 object the value of 20.5. Here is a complete program that uses the **Triangle** class.

Program 3.2: Area.java

```
class Triangle
{
    double base, height;
    void readData(double b, double h)    //definition of method
    {
            base=b;
            height=h;
    }
    double findArea( )//definition of another method
    {
                double a = 0.5*(base *height);
                return a;
    }
}
class Area    //class with the main() method
{
    public static void main(String args[ ])
    {
        double area1, area2;
        Triangle t1=new Triangle();
        Triangle t2=new Triangle();
        t1.base=20.5;
        t1.height=15.5;
        area1= t1.base  * t1.height;
        t2.readData(30.0,15.0);
```

```
        area2=t2.findArea();
        System.out.println("Area of the first triangle is =
"+area1);
        System.out.println("\nArea of the second triangle is =
"+area2);
    }
}
```

The *Triangle* class has no *main()* method. So it cannot be executed as Java program. We need a separate class that does have a *main()* method. In the above program, the class *Area* contains the *main()* method. The name of the file where we write the program should have the name ***Area.java*** . If we execute the program with the above data, the output will look like:



If we write these two classes ***Triangle*** and ***Area*** into two separate files namely ***Triangle.java*** and ***Area.java***, then these two files should be saved in the same folder in the computer. To compile ***Area.java*** program, we must first compile the ***Triangle*** class as:

javac Triangle.java

This will create the bytecode file ***Triangle.class*** in the same folder. Now, we can compile and execute the ***Area.java*** file with the following statements: **javac Area.java**

java Area

If we look at the contents of the folder, we will find four files: the source code and bytecode for Triangle and the source code and bytecode for Area. The statements

```
Triangle t1=new Triangle();
Triangle t2=new Triangle();
```

will create two Triangle objects t1 and t2 in memory, which reserves memory for two base fields , two height fields, two readData() methods and two findArea() methods. They are distinguished by which reference we use. Each of the fields is initialized in both objects using the dot operator with the following statements:

```
t1.base=20.5;
t1.height=15.5;
```

The Triangle reference t1 points to the Triangle object whose base is 20.5 and height is 15.5. The base and height field of t2 object is initialized to 30.0 and 15.0 with the statement
t2.readData(30.0,15.0);

To compute the area of the triangle t1, we can use any one of the following statements:

```
area1= t1.base  * t1.height;
area1=t1.findArea();
```

Similarly, for t2, we can use

```
area2=t2.base  * t2.height;
area2=t2.findArea();
```

3.4 CONSTRUCTORS

It would be simpler and more concise to initialize an object when it is first created. Java supports constructors that enable an object to initialize itself when it is created. This section provides a brief introduction about the **Constructor** and how constructors are overloaded in Java.

Constructor is always called by ***new*** operator. Constructors are declared just like as we declare methods, except that the constructor does not have any return type. The name of the constructors must be

the same with the class name where it is declared. It is called when a new class instance is created, which gives the class an opportunity to set up the object for use. Constructors, like other methods, can accept arguments and can be overloaded but they cannot be inherited. For example, let us consider the same *Triangle* class. We can replace the *readData()* method by a constructor. This can be accomplished as follows:

Program 3.3: TriangleArea.java

```
class Triangle
{
    double base, height;
     Triangle(double b, double h) //constructor with two arguments
        {
                base=b;
                height=h;
         }
        double findArea( )    //definition of method findArea()
        {
                double area = 0.5*(base *height);
                return area;
        }
}
class TriangleArea    //class with the main() method
{
    public static void main(String args[ ])
    {
        Triangle t1=new Triangle(20.5, 15.5);  // call of constructor
              double area1= t1.findArea();
              System.out.println("Area of the triangle is =  "+area1);
     }
}
```

The output of the program will be:

Area of the triangle is 317.75

The *Triangle* class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The constructor in the Triangle class takes two arguments. The statement Triangle t1= new Triangle(20.5, 15.5); provides 20.5 and 15.5 as values for those arguments.

3.4.1 DEFAULT AND COPY CONSTRUCTOR

There are two special kinds of constrcutors that a class may have: a *default constructor* and a *copy constructor*.

The ***default constructor*** is what gets called whenever we create an object by calling its constructor with no arguments. If we donnot define a constructor for a class, a *default constructor* is automatically created by the compiler. It initializes all instance variables to default value(zero for numeric types, null for object references, and false for booleans). There may be only one default constructor in a class.

The ***copy constructor*** is a constructor whose only parameter is a reference to an object of the same class to which the constructor belongs. It is called copy constructor as it is used to duplicate an existing object of the class.

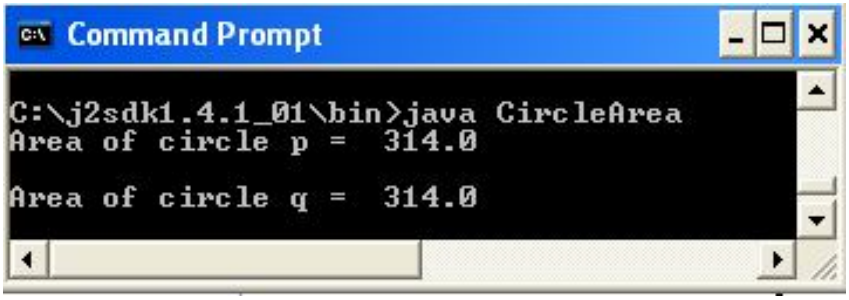
Let us consider another example to illustrate these two constructors:

Program 3.4: CircleArea.java

```
class Circle
{
        double radius;
        Circle()                        //default constructor
        {
                radius =10.0;
```

```
        }
        Circle(Circle c)          //copy constructor
        {
                radius = c.radius;
        }
        double findArea( )        //for calculation of area
        {
                double area = 3.14 *radius *radius;
                return area;
        }
}
class CircleArea    //class with the main() method
{
        public static void main(String args[ ])
        {
 double p_area,q_area;
 Circle p = new Circle();      //invokes the default constructor
 Circle q = new Circle(p);     //invokes copy constructor
              p_area= p.findArea();
              q_area =q.findArea();
          System.out.println("Area of circle p =  "+p_area);
          System.out.println("\nArea of circle q =  "+q_area);
        }
}
```

The output of the above program will be like this:



The statement **Circle p = new Circle();** invokes the *default constructor* and it creates a Circle object **p** with radius 10.0. The statement Circle **q = new Circle(p);** invokes the *copy constructor* and it also creates another Circle object q with the same radius value 10.0. The object p and q are two separate but equal objects. In the output we can see that the area of both the circles are same. All fields in a class will automatically be initialized with their type's default values unless the constructor explitcitly uses other values.

3.4.2 OVERLOADING OF CONSTRUCTORS

Overloading of constructors means multiple constructors in a single class. *Program 3.4* is also an example of constructor overloading as there are two constructors in that program. Constructor can be overloaded provided they should have different arguments because Java compiler differentiates constructors on the basis of arguments passed in the constructor.

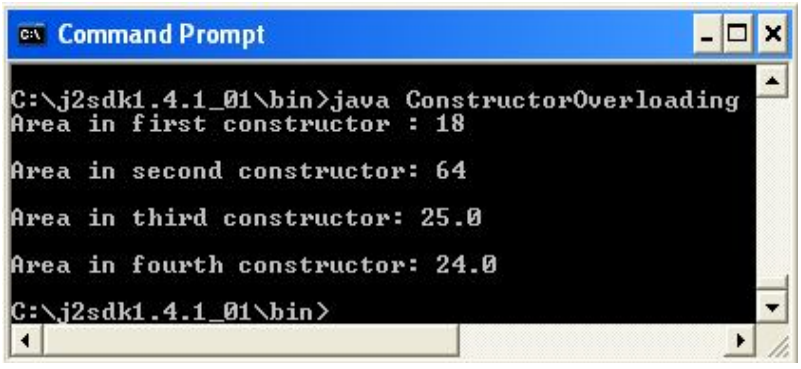
Let us consider the following *Rectangle* class for the demonstration of constructor overloading. After you going through it the concept of constructor overloading will be more clear. to you. In the example below we have written four constructors each having different arguments types.

Program 3.5: ConstructorOverloading.java

```
class Rectangle
{
    int l, b;  // for int type length and breadth
    float p, q;   // for float type length and breadth
    Rectangle(int x, int y)   // two int type argument constructor
            {
                    l = x;
                    b = y;
            }
            int first() {                   //method
                        return(l * b);
             }
    Rectangle(int x) { //one int type argument constructor
                        l = x;
                        b = x;
            }
            int second() {          //method
                       return(l * b);
            }
    Rectangle(float x) {    //one float type argument constructor
                        p = x;
                        q = x;
           }
           float third()
           {
                    return(p * q);
            }
    Rectangle(float x, float y) {
                    p = x;
                    q = y;
```

```
        }
        float fourth()    {
                return(p * q);
        }
}
class ConstructorOverloading  {
        public static void main(String args[ ])        {
Rectangle r1=new Rectangle(2,4);
int area1=r1.first();
System.out.println(" Area in first constructor : " + area1);
Rectangle r2=new Rectangle(5);
int area2=r2.second();
System.out.println("\nArea in second constructor: " + area2);
Rectangle r3=new Rectangle(2.0f);
float area3=r3.third();
System.out.println("\nArea in third constructor: " + area3);
Rectangle r4=new Rectangle(3.0f,2.0f);
float area4=r4.fourth();
System.out.println("\nArea in fourth constructor: " + area4);
        }
}
```

The output will be like this:



Constructor can also invoke other constructors with the ***this*** and ***super*** keywords. We will discuss the first case here, and return to that of the superclass constructor after we have talked

more about subclassing and inheritance. A constructor can invoke another, overloaded constructor in its class using the reference ***this()*** with appropriate arguments to select the desired constructor. If a constructor calls another constructor, it must do so as its first statement: ***this()*** calls another constructor in same class. Often a constructor with few parameters will call a constructor with more parameters, giving default values for the missing parameters. We can use *this* to call *other constructors* in the same class.

```
class Triangle
{
        double base, height;
        Triangle(double b,double h)          {
                base = b;
                height=h;
        }
        Triangle(double b)
        {
                this(b, 20.50 );
        }
}
```

In the above code, the class Triangle has two constructors. The first, accepts arguments specifying the triangles's base and height. The second constructor takes just the base as an argument and, in turn, calls the first constructor with a default value 20.50 for height. We have considered a simple example for clear understanding but the advantage of this approach is that we can have a single constructor do all the complicated setup work; other auxiliary constructors simply feed the appropriate arguments to that constructor. The call to ***this()*** must appear as the first statement in our second constructor. It should be remembered that we can invoke a second constructor only as the first statement of another constructor.

3.5 PASSING ARGUMENTS TO METHODS

In Java, we can pass an argument of any valid Java data type into a method similar to functions in other programming languages like C, C++ etc. This includes simple data types such as characters, integers, floats, doubles and boolean as well as complex types such as arrays, objects etc. Arguments provide information to the method from outside the scope of the method. A method in Java always specifies a return type. The returned value can be a primitive type, a reference type, or the type *void* , which indicates no returned value.

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor.

For example, let us consider the following method that computes the area of a rectangle:

```
int computeArea(int width, int height)    //method header
{
        int area;                  // area is a local variable
        area = width * height;
        return area;
 }
```

The first *int* indicates that the value this method returns is going to be an integer. The name of the method is “computeArea”, and it has two integer parameters: *length* and *breadth*. The *body* of the method starts with the left brace, “{“ and end with the right brace, “}”. The method is returning the area with the *return* statement. For calling the method, we can write :

```
int a= r1.computeArea(16,8);        //method call
```

- **Formal and Actual parameter :**

We use the term *formal parameters* to refer to the parameters in the definition of the method. In the example, *width* and *height* are the formal parameters. We use the term *actual parameters* to refer to

variables in the method call. They are called “actual” because they determine the actual values that are sent to the method.

Let us consider what happens when we pass arguments to a method. In Java, all primitive data types (e.g., int, char, float) are *passed by value*. The reference types (i.e., any kind of object, including arrays and strings) are used through references. An important distinction is that the references themselves (the pointers to these objects) are actually primitive types and are passed by value too.

- **Pass-by-Value**

Pass-by-value means that when we call a method, a copy of the *value* of each actual parameter is passed to the method. We can change that copy inside the method, but this will have no effect on the actual parameter. Unlike many other languages, Java has no mechanism for changing the value of an actual parameter.

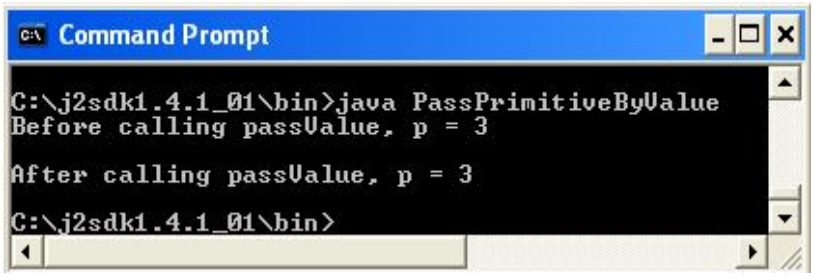
Java has eight primitive data types. Let us consider the following example for the demonstration of *pass-by-value*:

Program 3.6: PassPrimitiveByValue.java

```
class PassPrimitiveByValue
{
    public static void main(String[ ] args)
    {
        int p = 3;
        System.out.println("Before calling passValue, p = " + p);
        passValue(p);          //call passValue() with p as argument
        System.out.println("\nAfter calling passValue, p = " + p);
    }
```

```
    public static void passValue(int p) //passValue() definition to
    {                                   //change the value of parameter
                p = 10;
    }
}
```

When we run this program, the output will be:



Here, we can see that the outputs are same before and after calling the method ***passValue()***. When Java calls a method, it makes a copy of its actual parameters' values and sends the copies to the method where they become the values of the formal parameters. Then when the method returns, those copies are discarded and the actual parameters have remained unchanged.

Passing variables by value affords the programmer some safety. Methods cannot unintentoially modify a variable that is outside of its scope. However, we often want a method to be able to modify one or more of its arguments. In the ***passValue()*** method, the caller wants the method to change the value through its arguments. However, the method cannot modify its arguments, and, furthermore, a method can only return one value through its return value. So, it is necessary to learn how a method can return more than one value, or have an effect (modify some value) outside of its scope.

To allow a method to modify a argument, we must pass in an object. Objects in Java are also passed by value; however, the value of an object is a reference. So, the effect is that the object is ***passed in by reference***. When passing an argument by reference, the method gets a reference to the object. A reference to an object is the address of

the object in memory. Now, the local variable within the method is referring to the same memory location as the variable within the caller.

For example, if a parameter to a method is an object reference. We can manipulate the object in any way, but we cannot make the reference refer to a different object.

Program 3.7: Record.java (demonstration of passing object to method)

```
class Record
{
    int roll;
    String name;
    static void tryObject(Record r)  //parameter is an object
    {                                //reference
        r.roll = 1;
        r.name = "Anuj";
    }
    public static void main(String [] args)
    {
        Record obj = new Record();  //object obj of Record class
        obj.roll = 2;
        obj.name = "Rahul";
        System.out.println("Before calling tryObject(), the record is: "+
                                        obj.name + "  " + obj.roll);
        tryObject(obj);         //method call
        System.out.println("After calling tryObject(), the record is: " +
                                        obj.name + "  " + obj.roll);
    }
}
```

In the output, we observe different records before and after calling the method.

The first print statement displays "**Rahul 2**". The second print statement displays "**Anuj 1**". Thus the object has been changed in this case.

The reference to *obj* is the parameter to the method, so the method cannot be used to change that reference; i.e., it cannot make *obj* reference a different Record. But the method can use the reference to perform any allowed operation on the Record that it *already* references.

It is often not good programming style to change the values of instance variables outside an object. Normally, the object would have a method to set the values of its instance variables.

3.6 RECURSIVE METHOD

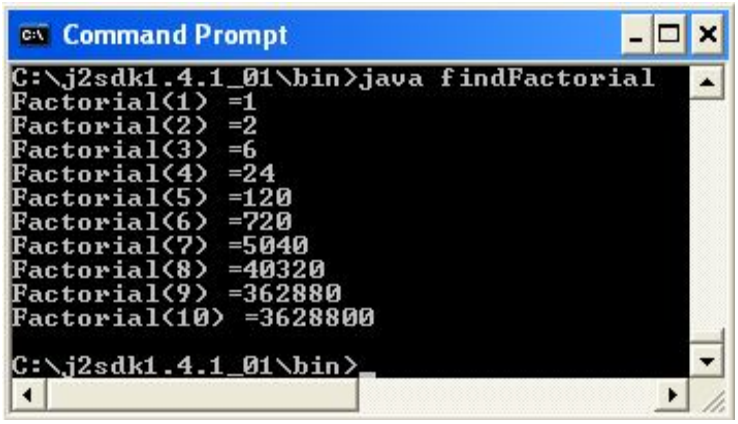
Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, *recursion* is the attribute that allows a method to call itself. A method that calls itself is said to be ***recursive method***.

One of the suitable examples of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. for example, 4 factorial is 1×2×3×4, or 24. Here is how a factorial can be computed by use of a recursive method.

Program 3.8 : findFactorial.java

```
class findFactorial
{
  public static void main(String [] args)
  {
    for(int i=1;i<=10;i++)
    System.out.println("Factorial("+ i +") =" + fact(i)); //method call
  }
  static long fact(int n)//method definition
  {
        if(n<2)
          return 1;
        return n*fact(n-1); //Recursive call
    }
}
```

The output will display the factorial of 1 to 10 as follows:



When **fact()** is called with an argument of 1, the function returns 1; otherwise it returns the product of **n* fact(n-1)**. To evaluate this expression, **fact()** is called with **n-1**.

The main advantage of recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

CHECK YOUR PROGRESS 1

1. Fill in the blanks :
   (i) In Java, data items are called_____.
   (ii) An _____ is defined to be the instance of a class.
   (iii) The extention of java file name should be _____
   (iv) Class containg the ***main( )*** method shold have the _____ name with the . file name with .java extension.
   (v) Java allocates storage for an object with the_____ operator.
   (vi) The _____ operator links the name of the object with the name of an instance variable or method.
   (vii) Contructors have no _____ .
   (viii) _____ calls another constructor in same class.
   (ix) In Java, all primitive data types are _____to a method.
   (x) _____ is the process of defining something in terms of itself
   (xi) Objects in Java are also passed _____.
   (xii) A copy of the value of each _____ parameter is passed to the method in case of pass by value.
   (xiii) A class can have only _____ default constructor.
   (xiv) A _____constructor is a constructor that replicates an existing object.
   (xv) Constructors are used to _____ an object of a class.

2. Write the method definition that implements the power function:
   static double Power (double x, int n)
   The method should return the value of x raised to the power n.
   For example Power(2.0,-3) would return 2^{-3}=0.125.

3. What is a constructor signature ?
   .....
   .....
   .....

3.7 INHERITANCE

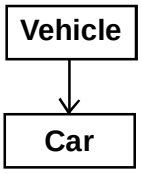
Reusability is one of the important feature of object-oriented-programming and it can be achieved through ***inheritance***. Java supports the concepts of inheritance. With the use of inheritance the information is made manageable in a hierarchical order. Inheritance can be defined as the process where one object acquires the properties of another. When we want to create a new class and there is already a class that includes some of the code that we want, we can derive the new class from the existing class. In doing this, we can reuse the fields and methods of the existing class without rewriting them again.

A class that is derived from another class is called a ***subclass*** (also a *derived class*, *extended class*, or *child class*). A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. The class from which the subclass is derived is called a ***superclass*** (also a *base class* or a *parent class*). Constructors cannot be inherited by subclasses, but the constructor of the superclass can be invoked from the subclass. In Java, inheritance is implemented by the process of extension. To define a new class as an extension of an existing class, we simply use an ***extend*** clause in the header mof the new classes definition. The concept of inheritance is used to make the things from general to more specific.

For example, when we hear the word 'vehicle' then we get an image in our mind that it moves from one place to another and that is used for traveling or carrying goods but the word vehicle does not specify whether it is two or three or four wheeler because it is a general word. But the word car makes a more specific image in mind than vehicle, that the car has four wheels . It concludes from the example that car is a specific word and vehicle is the general word. If we think technically about this example then vehicle is the super class (or base class or parent class) and car is the subclass or child class because every car has the features of its parent (in this case vehicle) class. At this point, we are going to describe the types of inheritance supported by Java.

3.7.1 SINGLE INHERITANCE

When a subclass is derived from its parent class then this mechanism is known as single inheritance. In case of single inheritance there is only a sub class and its parent class. It is also called *one level* inheritance. The pictorial representation of single inheritance is as follows:



Single Inheritance

For example, let us consider a simple example for the demonstration of single inheritance:

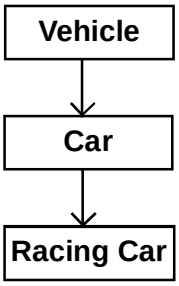
```
//Program 3.9 : B.java (Program showing Single Inheritance)
class A    //  super class A
{
        int x;
        int y;
        int getValue(int p, int q)        {
                x = p;
                y = q;
               return(0);
         }
        void Show()        {
                 System.out.println(x);
         }
}
class B extends A        // subclass B inheriting getValue A
{
     public static void main(String args[ ]) {
          A a = new A();
          a.getValue(5,10);
           a.Show();
```

```
    }
    void display()    {
        System.out.println("I am in B");
    }
}
```

The output will display only 5. The ***getValue()*** and ***show()*** are members of superclass ***A***. With the statement ***a.getValue(5,10);*** the ***getValue( )*** is inherited from class ***A***. As the ***Show()*** method of class ***A*** is displaying only the first parameter so it is displaying only one value in the subclass ***B*** although it is taking two values 5 and 10 when it invoked by the statement ***a.Show()*** in class subclass ***B*** .

3.7.2 MULTILEVEL INHERITANCE

It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as the *multilevel inheritance*. The derived class is called the subclass or child class for its parent class and this parent class works as the child class for its just above (parent) class. Multilevel inheritance can go up to any number of level. The pictorial representation of multilevel inheritance is as follows:



Multilevel Inheritance

// Program 3.10 : C.java (Program showing Mutilevel Inheritance)

```
class A
{
        int x;
```

```
        int y;
        int get(int p, int q)
        {
                x = p;
                y = q;
               return(0);
        }
       void show()
       {
              System.out.println(x);
       }
}
class B extends A      //subclass B inheriting from A
{
       void Showb()
        {
             System.out.println("I am in B ");
       }
}

class C extends B            //subclass C inheriting from B
{
       void Display()
       {
             System.out.println("I am in C");
       }
       public static void main(String args[])
      {
              A a = new A();
             a.get(5,10);
             a.show();
       }
}
```

The output of the above program will be 5. Here, ***a*** is an object of superclass ***A*** and it is inheriting ***get( )*** and ***show( )*** methods of ***A***. The subclass ***B*** has one method ***Showb()***. The class ***C*** is the subclass of ***B*** and it has one ***Display()*** method. We can also create objects of class ***B*** and ***C*** and use these two method ***Showb()*** and ***Display()*** for displaying the messages "***I am in B***" and "***I am in C***" respectively.

The mechanism of inheriting the features of more than one base class into a single class is known as *multiple inheritance*. ***Java does not support multiple inheritance.*** But the multiple inheritance can be achieved by using the ***interface***. In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class. The concept of interface will be discussed in the next unit of this block.

3.8 MODIFIERS

Modifiers are keywords that we add to those definitions to change their meanings. To use a modifier, we include its keyword in the definition of a class, method, or variable etc. The Java language has a wide variety of modifiers. These are listed in the following table:

Class Modeifier	Meaning
abstract	The class cannot be instantiated i.e., we cannot create objects of that class.
final	The class cannot be extended.
public	Its members can be accessed from any other class.

Field Modifier	Meaning
private	It is accessible only from within its own class.
protected	It is accessible only from within its own class and its extensions.
public	It is accessible from all classes.
final	It must be initialized and cannot be changed
transient	It is not part of persistent state of an object
volatile	It may be modified by asychronous threads

Method Modifier	Meaning
final	It cannot be overridden in class extensions
native	Its body is implemented in another programming language
private	It is accessible only from within its own class.
protected	It is accessible only from within its own class and its extensions.
public	It is accessible from all classes.
static	It has no implicit argument.
synchronized	It must be locked before it can be invoked by a thread.
volatile	It may be modified by asynchronous threads.

Constructor Modifier	Meaning
private	It is accessible.
protected	It is accessible only from within its own class and it extensions.
public	It is accessible from all classes.

Local Variable Modifier	Meaning
final	It must be initialized and cannot be changed.

Table: 3.1: List of Modifiers

Some of the modifiers listed in the *table 3.1* are used to set access levels in the declaration of classes, fields, methods, constrctors and local variables etc. These are termed as ***access control modifiers*** which include :

private (Visible to the class only)

public (Visible to all)

protected (Visible to the package and all subclasses).

If none of these three is specified, i.e., the default, then the entity (class, field, constructor or method) has *package access*, which means that it can be accessed from any class in the same package.

- ***private***

Private access modifier is the most restrictive access level. Variables, methods and constructors that are declared private can only be accessed within the declared class itself. Class and interfaces cannot be private.

- ***public***

A class, method, constructor, interface etc. declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any other class. However, if the public class we are trying to access is in a different package, then the public class still need to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses. For example, The ***main()*** method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class. Thus we write:

```
public static void main(String[ ] arguments)
{
        // statements
}
```

- ***protected***

Variables, methods and constructors which are declared *protected* in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

To achieve another functionality Java provides a number of ***non-access modifiers*** which include *static, final, abstract, synchronized, volatile.*

- **static**

The modifier *static i*s used to specify that a method is a class method. Without it, the method is an instant method. An *instance method* is a method that can be invoked only when it is bound to an object of the class. A class method (also called static method) is a method that is invoked without being bound to any specific object of the class.

- **final**

The modifier *final* has differnent role depending upon which kind of entity it modifies. If it modifies a class, it indicates that class cannot have subclasses. If it modifies a field or local variable, it means that the variable must be initialized and cannot be changed (i.e., it is a constant).

We will learn how to use the modifiers like *abstract*, *transient*, *volatile*, *native*, *synthonized* etc. while describing diffiernet programs through out this course.

3.9 FINAL KEYWORD

The final keyword is used in several different contexts. Some of them are described below :

final classes

Sometimes we may like to prevent a class from being further sub-classes for security reason. If we write the ***final*** keyword infront of a class name in the class declaration then that class cannot be inherited i.e., we cannot create sub classes from that class. This is written as follows:

```
final class  A     // A cannot be extended further
{
        //statements
}
```

final methods

We can also declare a method with the final keyword. A method that is declared final cannot be overridden in a subclass. The main reason for declaring a method to be final is to guarabtee that it cannot be changed. For this, we have to just put keyword *final* after the access specifier and before the return type like this:

```
public final String convertText()
        {
                //statements
        }
```

final fields

We may also declare fields to be *final*. This is not the same thing as declaring a method or class to be final. When a field is declared final, it is a constant which will not and cannot change. It can be set once (for instance when the object is constructed, but it cannot be changed after that). Attempts to change it will generate a compile-time error.

final arguments

We can declare a method arguments as *final*. This means that the method will not directly change them. Since all arguments are passed by value, this is not absolutely required, but it is occasionally helpful.

3.10 ABSTRACT CLASS AND METHOD

An ***abstract class*** is a class that has atleast one *abstract method*. An abstract class cannot be instantiated. An ***abstract method*** is not actually implemented in the class. It is merely declared there. The body of the method is then implemented in subclasses of that class. . Java allows classes and methods declared to be abstract by means of *abstract modifier*.

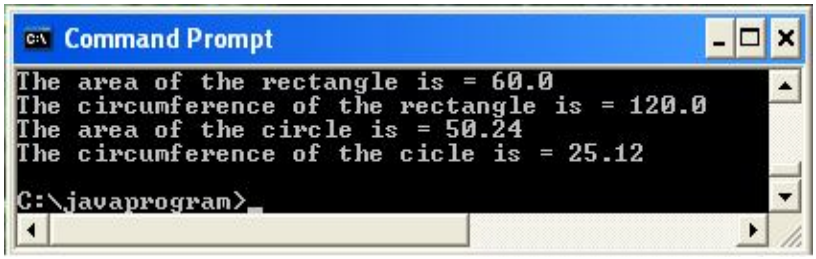
Let us consider the following example for the demonstration of abstract class and method. The program defines one abstract class ***Shape*** and two general classes ***Circle*** and ***Rectangle***.

```
//Program 3.11: DemoAbstract.java
abstract class Shape      //abstract class
{
        abstract double Area();         //abstract method
        abstract double Circumference();  //abstract method
}
//The abstract class Shape has two abstract methods: Area() and
//Circumference(). As abstract methods, they are declared with
//only their prototypes.

class Circle extends Shape          //derive class Circle
{
         double radius;          //field

        Circle(double radius)           //constructor
         {
                this.radius=radius;
          }
        double Circumference()          //method
        {
                return 2*3.14*radius;
        }
```

```
        double Area()                   //method
        {
                return 3.14*radius*radius;
        }
}
//The Circle class has one fields, one constructor and two methods.
//The methods Circumference() and Area() implement the corresponding
// abstract methods declared in the superclass Shape.

class Rectangle extends Shape       //Derive class Rectangle
{
        double length;      //field for specifying length
        double breadth;      //field for specifying breadth
        Rectangle(double length, double breadth)      //constructor
         {
                this.length=length;
                this.breadth=breadth;
         }
         double Area()                  //method
         {
                 return length*breadth;
         }
         double Circumference()      //method
        {
                return 2*length*breadth;
        }
}
//the Rectangle class has two fields specifying length and breadth,
//one constructor and two methods.The two methods Circumference()
// and Area() implement the corresponding abstract methods
// declared in the superclass Shape.
class DemoAbstract
{
        public static void main(String[] args)
```

```
    {
            Rectangle r = new Rectangle(10.0,6.0);
            Circle c = new Circle(4.0);
      System.out.println("The area of the rectangle is = "+ r.Area());
      System.out.println("The circumference of the rectangle is = "+
                        r.Circumference());
      System.out.println("The area of the circle is = "+ c.Area());
      System.out.println("The circumference of the cicle is = "+
                        c.Circumference());
    }
}
```

The ouput will be like this:



In the program, the abstract ***Area()*** method is declared in the ***Shape*** class above, because we want every subclass to have complete method that returns the areas of its instances and we want all those methods to have the same signature ***double Area()***. Similary for the abstract ***Circumference()*** method.

An abstract method is one that is intended to be *overridden* in each subclass. The abstract method specifies what its subclasses have to implement, but leaves the actual implementation up to them. Thus an abstract method can be regarded as an outline or a specification contact.

Method Overriding : *A method is overridden when another method with the same signature is declared in a subclass.*

3.11 STATIC MEMBERS

There may be some situation where we may want to define a class member that will be used independently without any object of that class. Generally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member

that can be used by itself, without reference to a specific instance. To create such a member, we have to precede its declaration with the keyword ***static***. When a member is declared *static*, it can be accessed before any objects of its class are created, and without reference to any object. We can declare both methods and variables to be *static*.

Static methods (***class methods***), like static variables, belong to the class and not to an individual instance of the class. It can be invoked by name, through the class name, without any objects around. Because it is not bound to a particular object instance, a static method can directly access only other static members of the class. It cannot directly see any instance variables or call any instance methods.

The most common example of a *static* member is ***main( )***. Method **main( )** is declared as *static* because it must be called before any objects exist. Instance variables declared as *static* are, essentially, global variables. When objects of its class are declared, no copy of a *static* variable is made. Instead, all instances of the class share the same *static* variable. Methods declared as *static* have several restrictions:

- They can only call other *static* methods.
- They must only access *static* data.

CHECK YOUR PROGRESS 2

1. What is the difference between public member and private member of a class ?

   .....

   .....

   .....

2. What is the difference between class method and an instance method.

   .....

.....

.....

3. State *True* or *False :*

(i) A static (class) method is a method that is invoked without being bound to any specific object of the class.

(ii) Protected members are accessible only from within its own class and its extensions.

(iii) Private members are accessible only from within its own class.

(iv) It is possible to instantiate an abstract class.

(v) An abstract method may must be part of an abstract class.

(vi) Final classes cannot be inherited.

(vii) An abstract class is a class that has atleast one abstract method.

(viii) Public members are not accesiible by all classes.

3.12 LET US SUM UP

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

- Java programs are organized by classes, which specify the behaviour of of objects, which control the actions of the program.
- Classes, objects and methods are basic components used in Java programming. A class can contain methods, fields, initialization codes etc. It serves as a blueprint for making class instances, which are runtime objects that implement the class structure.
- A Java program is a collection of one or more text files that define Java classes, atleast one of which is public and contains a *main()* method that has this specific form:

```
public static void main(String[ ] args)
{
        //statements
}
```

- A class definition specifies the variables and methods that are members of the class.
- Each class must be saved in a file with the same name as the class, and with the extension .java.
- A method is a sequence of declaration and executable statements that performs some individual task just like functions in C/C++.
- Method definition have some basic parts like *return type, name of the method, parameter list(*optional), *return statement*(if the method returns a value). If the method is not returning any value then the return type will be *void*.
- A method is said to be *recursive* when it calls itself.
- In Java, we can pass arguments to a method. Parameters which appear in the method call are *actual* parameters and the parameters which appear in the method definition are termed as *formal* parameters.
- The primitive types (e.g., byte, short, int, long, char, float, double, boolean) are passed by value to a method.
- Reference to an object can also be passed to a method as argument.
- Constructors are used to create new objects. Constructors must have the same name as that of the class when it is declared. It has no return type. Constructors are invoked with the *new* operator The *this* keyword calls another constructor in same class.
- Constructor with no argument is termed as *default constructor*. It is special as because if there is no other constructor in the class, the compiler will automatically define a public default constructor and initializes all fields to their type's default values.

- The *copy constructor* is one whose only parameter is a reference to an object of the same class to which the constructor belongs. It is usually used to duplicate an existing object of the class.
- When there are more than one constructor in a class then it is said to be constructor *overloading*. The name of all constructors should be same with the class name to which they belong but they should have different types and number of arguments.
- We can define one class based on another. This is called class *inheritance*. The base class is called a *superclass* and the derived class is called a *subclass*. A superclass can also be a subclass of another superclass.
- The first statement in the body of a constructor for a subclass should call a constructor for the superclass. If it does not, the compiler will insert a call for the default constructor for the superclass.
- Modifiers are some keywords which are used in the declaration of classes, fields, methods, constructors local variables etc. The use of different types of modifiers like *public, private, protected, final, abstract, static* etc. are described in this unit.
- Methods that are specified as static can be called even if no class objects exist, but a static method cannot refer to instance variables.
- An *abstract method* is a method that has no body defined for it, and is declared using the keyword abstract.
- An *abstract class* is a class that contains one or more abstract methods. It must be defined with the attribute abstract.

3.13 ANSWERS TO CHECK YOUR YOUR PROGRESS

Check Your Progress -1

1. (i) fields (ii) instance (iii) .java (iv) same (v) new (vi) dot (vii) return type (viii) this() (ix) passed by value (x) Recursion (xi) by value (xii) actual (xiii) one (xiv) copy (xv) initialize

2.

```
static double power(double x, int n)
{
        int i;
        double p=1.0;
        for(i=0; i<n; i++)
                p = p*x;
        for(i=0; i<-n; i++)
                p=p/x;
        return p;
}
```

3. A constructor signature consists of the constructor's name, number of parameters and type of parameter. The constructor signature must be unique when constructors are overloaded.

Check Your Progress - 2

1. A public class member can be accessed from methods of other classes. A private class member can be accessed only from methods of the class.

2. A ***class method*** is declared *static* and is invoked using the class name. For example,

```
double p = Math.abs(q);
```

invokes the class ***abs()*** that is defined in the ***Math*** class. An ***instance method*** is declared without the static modifier and is invoked using the name of the object to which it is bound. For example,

```
double p = obj. nextDouble();
```

invokes the class method *nextDouble()* that is defined in the *Random* class and is bound to the object *obj* which is an instance of that class.

3. (i) True (ii) True (iii) True (iv) False
   (v)True (vi) True (vii) True (viii) False

3.14 FURTHER READINGS

1. "*The Complete Reference -Java 2* " by Herb Hchildt, McGraw-Hill
2. "*JAVA How to Program*", Deitel & Deital, PHI Publication
3. "*Programming with JAVA- a Primer* ", E.Balagurusamy, TATA McGRAW Hill Publication.

3.15 MODEL QUESTIONS

1. What is a class? How does it accomplish data hiding?
2. What are objects? How are they created?
3. What is a constructor? How do we invoke a constructor? What are its special properties?
4. What is recursive method? Explain with example.
5. What is inheritance and how does it help us creating new classes quickly?
6. When do we declare a method or class final?
7. When do we declare a method or class abstract?
8. Discuss different levels of access protection available in Java.
9. What is inheritance? Explain single inheritance with an appropriate program.
10. What do you mean by abstract class? Write down a suitable example showing abstract class and abstract methods.

11. Design a class to represent a bank account. Include the following data members: *Annount number, Depositor name, Account type, Balance amount in the account* .

Methods: To assign initial value
To deposit an amount
To withdraw an amount after checking balance
To display depositor name and balance

*****

UNIT 4: ARRAYS, STRINGS AND VECTORS

UNIT STRUCTURE

4.1 LEARNING OBJECTIVES

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

- declare and initialize arrays for Java programming
- know the method *arraycopy ()*
- define multi-dimensional arrays
- describe strings
- illustrate the vectors

4.2 INTRODUCTION

In the previous unit, we have discussed the object-oriented nature of Java. We are already familiar with the array data structure. Array is a linear and homogenous data structure. In this unit, we will discuss the arrays, how the arrays are declared and initialized in Java language. In addition, the discussion of the strings and vectors are also covered in this unit.

4.3 ARRAYS

With the basic built-in Java data types, each identifier corresponds to a single variable. But when you want to handle sets of values of the same type, say, the first 1000 primes for example – you really don't want to have to name them individually. What you need is an **array**.

An array is a named set of variables of the same type. Each variable in the array is called an **array element**. To refer to a particular element in an array you use the array name combined with an integer value of type ***int***, called an **index**. You are already familiar with how to declare an array in C or C++. In Java, it is slightly different from C/C++. There are three steps to create an array in Java :

a) Declare a variable to hold array

b) Create memory locations

c) Put values into the memory locations

4.3.1 DECLARING ARRAY VARIABLES

You are not obliged to create the array itself when you declare the array variable. The array variable is distinct from the array itself. There are two ways to declare the array variables in Java:

data type arrayname[ ];

or

data type[ ] arrayname;

For example,

```
int sum[ ];
float percentage[ ];
int roll_number[ ];
int primes[ ];
```

The above declaration can be written as :

```
int[ ] sum;
float[ ] percentage;
int[ ] roll_number;
int[ ] primes;
```

Remember that in such types of declarations no memory has been allocated to store the array itself and the number of elements has not been defined.

4.3.2 CREATING AN ARRAY

Creating an array means allocating memory for it. Java allows us to create arrays using ***new*** operator. The syntax is given below :

arrayname = new type[size];

For example,

```
sum = new int[4];
percentage = new float[10];
roll_number = new float[15];
primes = new int[10];
```

The keyword **new** indicates that you are allocating new memory for the array, and int[4] specifies you want capacity for 4 variables of type int in the array. Since each element in the **sum** array is an integer variable requiring 4 bytes, the whole array will occupy 16 bytes, plus 4 bytes to store the reference to the array.

When an array is created like this, all the array elements are initialized to a default value automatically. Similarly, for the percentage, roll_number and primes array also a fixed size will be allocated. The above declared and created arrays are **one-dimensional array** since each of its elements is referenced using one index.

The following figure depicts the meaning of the declaring and creating an integer array named **sum**.

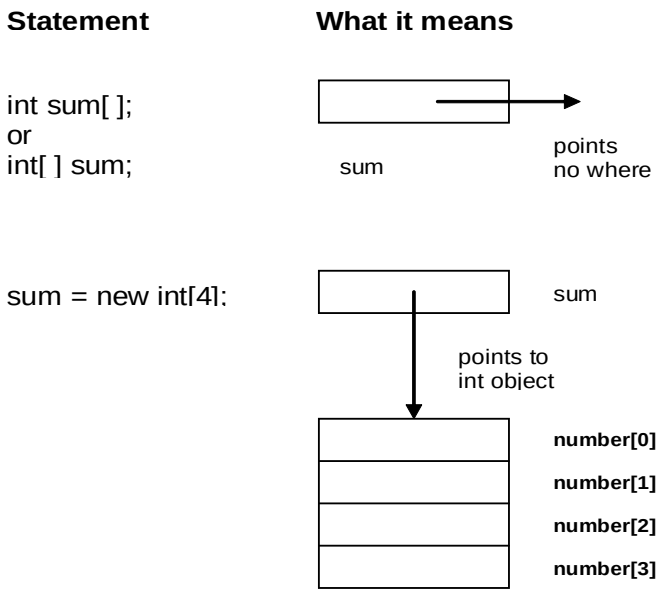


Fig. 4.1 Allocation of memory for an array

4.3.3 INITIALIZING ARRAYS

You can initialize an array with your own values when you declare it, and at the same time determine how many elements it will have. Following the declaration of the array variable, simply add an equal sign followed by the list of element values enclosed between braces.

For example,

```
int[ ] primes = {2, 3, 5, 7, 11, 13, 17};
```

Here, ***primes*** is an array of 7 elements. The array size is determined by the number of initial values; so no other statement is necessary to define the array.

If you specify initializing values for an array, you must include values for *all* the elements. If you only want to set some of the

array elements to values explicitly, you should use an assignment statement for each element.

For example:

```
int[ ] primes = new int[100];
primes[0] = 2;
primes[1] = 3;
```

The first statement declares and defines an integer array of 100 elements, all of which will be initialized to zero. The two assignment statements then set values for the first two array elements.

You can also initialize an array with an existing array. For example,

```
int[ ] even = {4, 6, 8, 10};
int[ ] number = even;
```

Here, both arrays refer to the same set of elements and you can access the elements of the array through either variable name – for example, even[2] refers to the same variable as number[2].

The following program demonstrates the initialization of one dimensional array :

Program 1: To find the average of given list of numbers.

```
class Average
  {
      public static void main(String args[])
          {
 double nums[] = {50.1, 51.2, 52.3, 53.4, 54.5, 55.6, 56.7, 57.8,
                        58.9, 59.0};
  double result = 0;
  int i;
  for(i=0; i<10; i++)
```

```
    result = result + nums[i];
    System.out.println("Average is " + result / 10);
}
    }
```

Output :

```
Average is 54.95
```

Array Length

You can refer to the length of the array using **length**, a data member of the array object. In our array example i.e. the array **sum,** its length can e assigned to an another variable as follows:

int size = sum.length

The following program 2 & 3 demonstrates the use of length object for computing the largest and the smallest number from a given list of numbers.

Program 2: To find the smallest and largest number from a given list of number.

```
class FindNumber
      {

   public static void main(String[] args)
     {

        //array of 10 numbers
     int numbers[] = new int[]{12,83,50,18,22,72,41,29,43,21};

        //assign first element of an array to largest and smallest
        int smallest = numbers[0];
        int largetst = numbers[0];
```

```
        for(int i=1; i< numbers.length; i++)
          {
                if(numbers[i] > largetst)
                        largetst = numbers[i];
                else if (numbers[i] < smallest)
                        smallest = numbers[i];

           }

        System.out.println("Largest Number is    : " + largetst);
        System.out.println("Smallest Number is : " + smallest);
         }
}
```

Output :

```
Largest Number is     : 83
Smallest Number is   : 12
```

Program 3 : To check whether a given number is palindrome or not.

```
class Palindrome
  {

      public static void main(String[] args)
        {

        //array of numbers
 int numbers[] = new int[]{2332,42,11,223,24};

        //iterate through the numbers
 for(int i=0; i < numbers.length; i++)
 {
```

```
        int number = numbers[i];
        int reversedNumber  = 0;
        int temp=0;

        /*
        * If the number is equal to it's reversed number, then
        * the given number is a palindrome number.
        * 121 is a palindrome number while 12 is not.
                */

        //reverse the number
       while(number > 0)
         {
        temp = number % 10;
        number = number / 10;
        reversedNumber = reversedNumber * 10 + temp;
          }

       if(numbers[i] == reversedNumber)
 System.out.println(numbers[i] + " is a palindrome number");
       else
System.out.println(numbers[i] + " is not a palindrome number");
 }

       }
 }
```

Output :

```
2332 is a palindrome number
42 is not a palindrome number
11 is a palindrome number
223 is not a palindrome number
24 is not a palindrome number
```

Reusing Array Variables :

you can use an array variable to reference different arrays at different points in your program. Suppose you have declared and defined the variable primes as before:

```
int[ ] primes = new int[10];
                    // Allocate an array of 10 integer elements
```

This produces an array of 10 elements of type *int*. Perhaps you want the array variable primes to refer to a larger array, with 50 elements say. You would simply write :

```
primes = new int[50];
                    // Allocate an array of 50 integer elements
```

Now the variable primes refer to a new array of values of type *int* that is entirely separate from the original. When this statement is executed, the previous array of 10 elements is discarded, along with all the data values you may have stored in it. The variable primes can now only be used to reference elements of the new array.

4.4 MULTIDIMENSIONAL ARRAYS

A table organized in rows and columns is a very effective means for communicating many different types of information. In Java, we represent table as two-dimensional arrays. The general syntax for allocating a two dimensional array is :

type [ ][ ] arrayname = new type[rows][cols];

For example,

```
int [ ][ ] matrix = new int [3][5];
```

This creates a two dimensional matrix having 3 rows and 5 columns that can store 15 integer values. We are already familiar with C or C++ how to access the elements of a two dimensional array. In Java

also, the same techniques are used. For example, to refer to the elements at the second column of the third row we will indicate it as:

matrix [2][1]

We can also initialize a two dimensional array as shown below :

```
int [ ] [ ] matrix = {
                     { 10, 5,  -3, 9, 2 },
                     {  1 , 0, 14, 5, 6 },
                     { -1, 7,  4,  9,  2 }
                       };
```

There is no limit to the number of dimensions an array can have. We can declare three-dimensional, four-dimensional, and higher dimensional arrays. However, arrays with a dimension higher than 2 are not frequently used in object-oriented languages.

The following figure depicts the representation of the matrix array in memory :

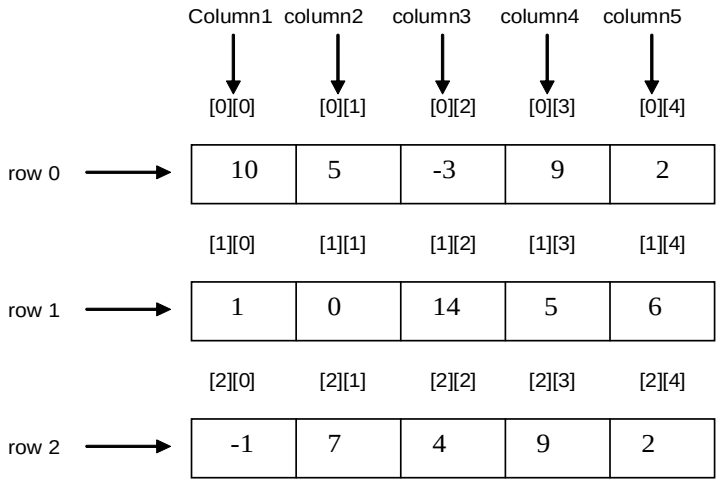


Fig. 4.2 Representation of matrix array

The following program demonstrate the use of two dimensional array in Java programming. This program will simply display the initialized list of elements as output.

Program 4 : To display the elements of a two dimensional array in matrix form.

```
class Matrix
    {
        public static void main(String args[])
         {
                int mat[][] = {
                                { 10, 5, -3, 9, 2 },
                                { 1, 0, 14, 5, 6 },
                                { -1, 7, 4, 9, 2 },
                                 };

                int i, j;
                for(i=0; i<3; i++)
                {
                    for(j=0; j<5; j++)
                    System.out.print(mat[i][j] + " ");
                     System.out.println();
                }
            }
         }
```

Output :

```
10,   5,   -3,   9, 2
 1,    0, 14,   5, 6
-1,    7,    4,   9, 2
```

Multidimensional arrays are actually arrays of arrays. It means we can create or allocate the sub arrays (columns). It is possible to create sub arrays of different lengths which will look like a triangle. For example, the following program creates a two dimensional array in which the sizes of the second dimension are unequal or the array looks like a triangle :

Program 5 : To display the elements of a two dimensional matrix in triangular form.

```
class TriangularArray
  {
     public static void main(String args[])
      {
        int arr[][] = new int[4][];
        int i;
        for(i=0; i<4; i++)       // Creates triangular array
        arr[i] = new int[i+1];
        int j, k = 0;
        for(i=0; i<4; i++)       // Assigns the elements
        for(j=0; j<i+1; j++)
          {
                arr[i][j] = k;
                k++;
           }
          for(i=0; i<4; i++)    // Display the elements
               {
                for(j=0; j<i+1; j++)
                System.out.print(arr[i][j] + " ");
                System.out.println();
                }
           }
     }
```

Output :

```
0
1 2
3 4 5
6 7 8 9
```

CHECK YOUR PROGRESS 1

1. Choose the correct answer from the following :

   (i) Which of the following statements about arrays are true?

   A. Arrays are a group of variables containing values that all have the same type.

   B. Elements are located by index or subscript.

   C. The length of an array c is determined by the expression c.length();.

   D. The zeroth element of array c is specified by c[0].

   a. A, C, D. b. A, B, D.

   c. C, D. d. A, B, C, D.

   (ii) Consider the code segment below. Which of the following statements is not true?

   int g[];

   g = new int[23];

   a. The first statement declares an array reference.

   b. The second statement creates the array.

   c. g is a reference to an array of integers.

   d. The value of g[3] is -1.

   (iii) Which of the following initializer lists would correctly set the elements of array n?

   a. int n[] = { 1, 2, 3, 4, 5 };.

   b. array n[int] = { 1, 2, 3, 4, 5 };.

   c. int n[5] = { 1; 2; 3; 4; 5 };.

   d. int n = new int(1, 2, 3, 4, 5);.

(iv) Consider the following Java Program :

```
public class Test
{
   public static void main( String args[] )
     {
int a[] = { 99, 22, 11, 3, 11, 55, 44, 88, 2, -3 };
int result = 0;
for ( int i = 0; i < a.length; i++ )
{
    if ( a[ i ] > 30 )
       result += a[ i ];
}
 System.out.println( "Result is: " +result );
      }
 }
```

The output of this Java program will be:

a. Result is: 280. b. Result is: 154.

c. Result is: 286. d. Result is: 332.

(v) Which statement below initializes array items to contain 3 rows and 2 columns?

a. int items[][] = { { 2, 4 }, { 6, 8 }, { 10, 12 } };.

b. int items[][] = { { 2, 6, 10 }, { 4, 8, 12 } };.

c. int items[][] = { 2, 4 }, { 6, 8 }, { 10, 12 };.

d. int items[][] = { 2, 6, 10 }, { 4, 8, 12 };.

(vi) 26. For the array in the previous question, what is the value returned by items[1][0]?

a. 4. b. 8. c. 12. d. 6.

(vii) Which of the following statements creates a multidimensional array with 3 rows, where the first row contains 1 value, the second row contains 4 items and the final row contains 2 items?

a. int items[][] = { { 1, null, null, null }, { 2, 3, 4, 5}, { 6, 7, null, null } };.

b. int items[][] = { { 1 }, { 2, 3, 4, 5 }, { 6, 7 } };.

c. int items[][] = { { 1 }, { 2, 3, 4, 5 }, { 6, 7}, {});.

d. int items[][] = { { 1 }, { 4 }, { 2 } };.

4.5 STRINGS

The strings are nothing but a sequence of characters. **In Java, strings are objects of the class *String***. The **String** and **StringBuffer** classes are standard class that comes with Java, and they are specifically designed for creating and processing strings. A sequence of characters separated by double quotes is called **String constants**. Java handles **String** constants in the same way that other computer languages handle “normal” strings. Objects of type **String** are immutable it means that once a **String** object is created, its contents cannot be altered.

As String is a class, we can create an instance or object and give it a name. For example,

```
String name;
name = new String(“KKHSOU”);
```

Unlike in other classes, the explicit use of **new** to create an instance or object is *optional* for the String class. So, we can create a new String object in this way :

```
String name;
name = “KKHSOU”;
```

or

```
String name = “KKHSOU”;
```

Once you have created a **String** object, you can use it any where that a string is allowed. For example, the following statement displays 'KKHSOU' :

```
System.out.println(name);
```

We are already using the ***length*** method which is under the String class to get the length of an array. In case of finding the length of a string also we can use it. For example,

```
int i = name.length();
```

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings. For example, this statement

```
String myString = "I" + " like " + "Java.";
```

results in **myString** containing "I like Java."

The following program demonstrates the above concept :

Program 6 : To concatenate two strings using the + operator.

```
class StringJoin
    {
        public static void main(String args[])
            {
                String str1 = "Krishna Kanta Handique";
                String str2 = "State Open University";
                                // str3 holds the resultant string
                String str3 = str1 + " and " + str2;
                System.out.println("\n First String :" +str1);
                System.out.println("\n Second String :" +str2);
                System.out.println("\n Resultant String :" +str3);
            }
    }
```

Output :

First String : Krishna Kanta Handique
Second String : State Open University
Resultant String : Krishna Kanta Handique State Open University

4.5.1 STRINGS METHODS

The String class defines a number of methods that allow us to accomplish a variety of string manipulation tasks. The following table lists some of the commonly used string methods.

Method Call	Task Performed
s2=s1.toLowerCase;	Converts the String s1 to all lowercase
s2=s1.toUpperCase;	Converts the String s1 to all Uppercase
s2=s1.replace('x','y');	replaces all appearances of x with y
s2=s1.trim();	removes white spaces at the begining and end of the String s1
s1.equals (s2)	Returns true if s1 equals to s2
s1.equalsIgnoreCase(s2)	Returns true if s1=s2, ignoring the case of Characters
s1.length()	gives the length of s1
s1.CharAt(n)	gives the nth character of s1
s1.compareTo(s2)	returns negative if s1<s2, positive if s1>s2, and zero if s1 is equal to s2
s1.concat (s2)	concatenates s1 and s2
s1.substring(n)	gives the substring starting from nth character
s1.substring(n,m)	gives the substring starting from nth character upto mth (not including mth)

The following program demonstrates the use of some string methods:

Program 7 : A string is given below :

BACHELOR OF COMPUTER APPLICATION FOURTH SEMESTER

Find the folllowing :

a) its length,

b) the character at index 6

c) convert it to lowercase letter

d) a substring from 2 to 15

e) length of the substring

f) the character at index 3

```
class TestString
  {
        public static void main(String[] args)
        {
        String text = " BACHELOR OF COMPUTER APPLICATION
                                FOURTH SEMESTER";
        System.out.println("\n\nThis string is :" +text);

        // calculates length of the string
        System.out.println("Length of the String is :" +text.length());
                        // finds the character at index 6 position
        System.out.println("The character at index 6 is :" +text.charAt(6));
                        // Converts the string into lowecase letter
        System.out.println("The string in lowercase letter is :"
                                                +text.toLowerCase());
                        // Finding the substring of the main string
        String sub = text.substring(2, 15);


        System.out.println("sub=text.substring(2, 15)) :" +sub);
                        // Length of the sub string
        System.out.println("sub.length :" +sub.length());
                        // character at position 3
        System.out.println("sub.charAt(3) :" +sub.charAt(3));

          }
   }
```

Output :

Fig. 4.1 Output of Program 7

You can also test two strings for equality by using the method **equals( )**. The general form of this method is **s1.equals (s2)** which means it will return true if s1 equals to s2.

The following program demonstrates the equality of two string.

Program 8 : To check equality of two strings.

```
class EqualString
    {
        public static void main(String args[])
          {
               String str1 = "BCA Second Semester";
               String str2 = "BCA Third Semester";
               String str3 = str1;              //str1 string assigns to str3

                                        // Finds the length of strings
              System.out.println("\nLength of str1: " + str1.length());
              System.out.println("\nLength of str2: " + str2.length());
              System.out.println("\nLength of str3: " + str3.length());
```

```
        if(str1.equals(str2))
                System.out.println("str1 == str2");
        else
                System.out.println("str1 != str2");

        if(str1.equals(str3))
                System.out.println("str1 == str3");
        else
                System.out.println("str1 != str3");
      }
}
```

Output :

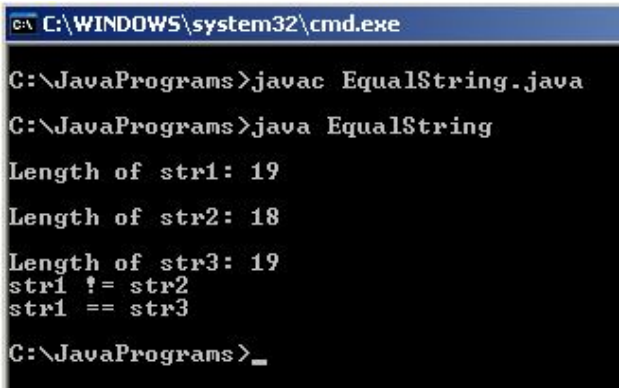


Fig 4.2 Output of Program 8

In the program, you see the equality operator "==" which is used to test whether two references refer to the same object. Here in our expression, if str1 and str2 are both String references, then the expression str1 == str2 will be true only if str1 and str2 are synonyms. Here in the example, str1 and str2 are not equivalent so it will return false and hence prints the else statement.

String Arrays :

We can also create and use arrays for storing strings. The following statement will create an string array named **arraySt** of size 5 to hold 5 string constant.

String arraySt[ ] = new String[5];

We can assign the strings to the **arraySt** element by element using five different statements or more efficiently using a **for** loop.

The following program demonstrates the use of the string array :

Program 9 :

```
class arrayStr
  {
        static String arraySt[ ] ={"BCA 1st Sem","BCA 2nd Sem","BCA
                                   3rd Sem","BCA 4th Sem","BCA
                                   5th Sem", "BCA 6th Sem"};

        public static void main(String args[])
           {
               for(int i=0;i<6;i++)
               {
                     System.out.println(arraySt[i]);
                }
             }
         }
```

Output :

```
BCA 1st Sem
BCA 2nd Sem
BCA 3rd Sem
BCA 4th Sem
BCA 5th Sem
BCA 6th Sem
```

4.5.2 StringBuffer Class

The string class is one of the most useful classes in Java. But its instances or objects are immutable i.e. they cannot be changed. In the above examples, whenever a string is modified, it has to be done by constructing a new String object, either explicitly or implicitly. Java provides another standard class for defining strings called **StringBuffer**, and its object can be altered directly. Strings that can be changed are often referred to as **mutable strings**.

We will use the StringBuffer objects when we are transforming strings i.e. adding, deleting, or replacing substrings in a string.

The following are the commonly used StringBuffer methods :

capacity()

Returns the current capacity of the String buffer.

length()

Returns the length (character count) of this string buffer.

toString()

Converts to a string representing the data in this string buffer

insert(int offset, char c)

Inserts the string representation of the char argument into this string buffer.

delete(int start, int end)

Removes the characters in a substring of this StringBuffer

replace(int start, int end, String str)

Replaces the characters in a substring of this StringBuffer with characters in the specified String.

reverse()

The character sequence contained in this string buffer is replaced by the reverse of the sequence.

append(String str)

Appends the string to this string buffer.

setLength(int newLength)

Sets the length of this String buffer.

The following program demonstrates the using of StringBuffer methods :

Program 10 :

```
public class FunctionStrbf
  {
    public static void main(String[] args)
     {
                                    //  Creation of Strings
 StringBuffer strBuf1 = new StringBuffer("KKHSOU");
 StringBuffer strBuf2 = new StringBuffer(40);  //With capacity 40
 StringBuffer strBuf3 = new StringBuffer();   //Default Capacity 16
 System.out.println("\n\n strBuf1 : " + strBuf1);
                          // Finds the capacity of StringBuffer
 System.out.println("strBuf1 capacity : " + strBuf1.capacity());
 System.out.println("strBuf2 capacity : " + strBuf2.capacity());
 System.out.println("strBuf3 capacity : " + strBuf3.capacity());
                          // Finds the length of Buffer1
 System.out.println("strBuf1 length : " + strBuf1.length());
                          // Finds the character at position 2
 System.out.println("strBuf1 charAt 2 : " + strBuf1.charAt(2));

                      // Appends a string to strBuf 3
 strBuf3.append("Fourth Semester BCA");

 System.out.println("strBuf3 when appended with a String : "
                          + strBuf3.toString());

 strBuf3.insert(16, 'M');          // insert 'M' at position 16
```

NOTE

Note that a vector can be declared without specifying any size explicitly. A vector can accomodate an unknown number of items. Even, when a size is specified, this can be overlooked and a different number of items may be put into the vector. Remember, in contrast, an array must always have its size specified.

```
System.out.println("strBuf3 when M is inserted at 16 : "
                        + strBuf3.toString());

strBuf3.delete(11, 's');   // delete the substring from 11 position

System.out.println("strBuf3 when s is deleted at 11 : "
                        + strBuf3.toString());
strBuf3.reverse();       // reverse the string constant of strBuf3
System.out.println("Reversed strBuf3 : " + strBuf 3);

strBuf2.setLength(12); //set the length of strBuf 2 to 12
strBuf2.append("java programs"); // appends the string to strBuf2
System.out.println("strBuf2 : " + strBuf2);
}
   }
```

Output :

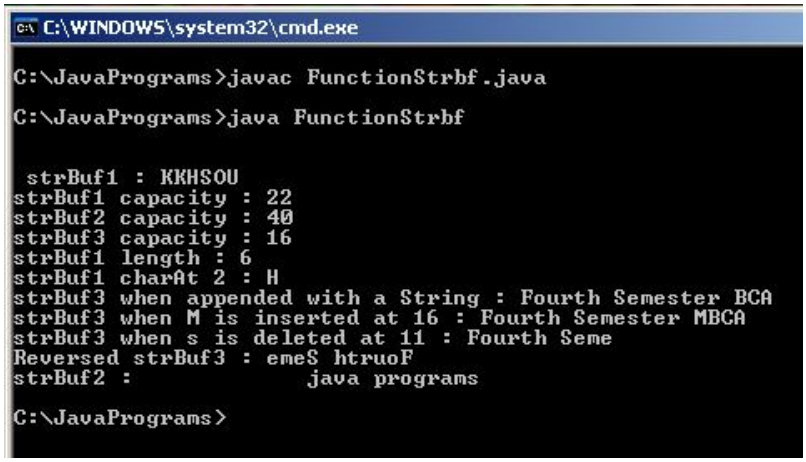


Fig. 4.3 Output of Program 10

4.6 VECTORS

You can consider a **Vector** as an expansible array. Before you can use an array, you need to know the maximum number of elements in order to declare it. If the array is full and it turns out that you need to add another element, you need to declare a bigger one, copy all the elements and add that element.

A Vector hides all that things. Internally, a Vector stores its elements also in an array but it will increase its capacity automatically to ensure all elements be stored. So, use a Vector when you don't know the number of elements ahead of time. A Vector is a standard class that resides in the **java.util package**.

The key difference between Arrays and Vectors in Java is that Vectors are dynamically-allocated. They aren't declared to contain a type of variable; instead, each **Vector contains a dynamic list of references or pointers to other objects**. Since Vector stores pointers to the objects, and not the objects themselves, these Vector items could be any kind of object. We can add and delete objects from the list as and when required. Always remember we can not directly store simple data types (int, float...etc) in vectors.

Arrays can be easily implemented as vectors. Vectors are created like arrays as follows :

Vector V = new Vector(); //declaring without size

or

Vector V = new Vector(3); // declaring with size

The vector class supports a number of methods that can be used to manipulate the vectors created. Important ones are listed below :

Method Call	**Task Performed**
v.addElement (item)	Add the item specified to the list at the end
v.elementAt(10)	Give the name of the 10th object
v.size()	Give the number of objects present
v.removeElement (item)	Removes the specified item from the list
v.removeAllElements()	Removes all the elements in the list
v.copyInto (array)	Copies all items from list to array
v.insertElementAt (item, n)	Inserts the item at n th position

Wrapper Classes

Always remember that vectors can not handle primitive data types like int, float, long, char and double. Primitive data types may be converted into object types by using the wrapper classes contained in the **java.lang** package. In the following shows the simple data types and their corresponding wrapper class types.

Wrapper Classes for Converting Simple Types

Simple Type	Wrapper Class
boolean	Boolean
char	Char
double	Double
float	Float
int	Int
long	Long

CHECK YOUR PROGRESS 2

(i) Character strings are represented by the class ___. (Fill in the blank)

(ii) An ___ is a container object that holds a fixed number of values of a single type. (Fill in the blank)

(iii) Consider the following string:

String strTest = "Programming in Java is easy**";**

a. What is the value displayed by the expression **strTest.length()**?

b. What is the value returned by the method call **strTest.charAt(12)**?

c. Write an expression that refers to the letter **J** in the string referred to by strTest.

(iv) Show two ways to concatenate the following two strings together to get the string "KKHSOU":

```
String str1= "KKH ";
String str2 = "SOU";
```

(v) What is the main difference between **String** class and **StringBuffer** class

.....

.....

.....

(vi) Which package of Java contains Vector Class?

.....

.....

.....

4.7 LET US SUM UP

1. An array holds multiple values of the same type, identified through a single variable name.
2. Individual elements of an array referred by using an index value of type int. The index value for an array element is the offset of that element from the first element in the array.
3. An array element can be used in the same way as a single variable of the same type.
4. You can obtain the number of elements in an array by using the length member of the array object.
5. A String object stores a fixed character string that cannot be changed. However, you can assign a given String variable to a different String object.
6. You can obtain the number of characters stored in a String object by using the *length()* method for the object.
7. The *String* class provides methods for joining, searching, and modifying strings – the modifications being achieved by creating a new String object.

8. A *StringBuffer* object can store a string of characters that can be modified.

9. You can change both the length and the capacity for a StringBuffer object.

10. Java does not support the concept of variable arguments to a function. This feature can be achieved in Java through the use of the Vector class contained in *java.util* package.

4.8 ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress -1

1. i) b ii) d iii) a iv) c v) a vi) d vii) b

Check Your Progress - 2

i) java.lang.String

ii) array

iii) a) 26

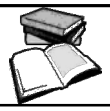
b) i

c) strTest.charAt(15)

iv) str1.concat(str2) and str1 + str2

v) Instance (objects) of the **String** class are immutable : they cannot be changed. Instances of the **StringBuffer** class donot have constraints i.e. they are mutable.

vi) **java.util** package



4.9 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

4.10 MODEL QUESTIONS

1. What is an array?
2. Write a statement to declare and instantiate an array to hold marks obtained by students in different subjects in a class. Make your own assumptions.
3. Write a program to add two matrices.
4. Write a program to extract a portion of a character string and print the extracted string.
5. What is a vector? How is it different from an array?
6. How does a String class differ from a StringBuffer class?
7. Write a program which will read a text and count all occurrences of a particular word.
8. Write a program to demonstrate the use of arraycopy() function.

******

UNIT 5: INTERFACES AND PACKAGES

UNIT STRUCTURE

5.1 LEARNING OBJECTIVES

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

- define an interface
- know the technique of extending an interface
- implement an interface
- illustrates the Java packages
- creating and accessing Java Packages

5.2 INTRODUCTION

So far, you have learned the object-oriented nature of Java. We have also discussed about the arrays, how the arrays are defined and also the String class, StringBuffer class for creating strings. Moreover, we have also came to know the vectors which are nothing but dynamic arrays.

In this unit, we will discuss two important concept of Java namely interfaces and packages. We will illustrate how to define interfaces and implement them. In addition, creating and accessing Java Packages are also covered in this unit.

5.3 INTERFACES

An interface is a collection of methods and variables like a class but it is not a class. An interface defines a set of methods but does not implement them. Writing an interface is similar to writing a class still there exists some differences. A class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements.

Definition

An interface is a named collection of method definitions (without implementations). An interface can also declare constants.

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Remember that - to implement an interface, a class must create the complete set of methods defined by the interface.

5.3.1 Defining an Interface

The syntax for defining an interface is very similar to that for defining a class, which is shown below :

```
interface interfaceName
    {
        return-type  methodName1(parameter-list);
        return-type  methodNname2(parameter-list);
        type  final-varname1 = value;
        type  final-varname2 = value;
                .....
                .....
        return-type  methodNameN(parameter-list);
        type  final-varnameN = value;
    }
```

From the above syntax we have seen that an interface definition has two components: **the interface declaration** and the **interface body**. The interface declaration declares various attributes about the interface, such as its name and whether it extends other interfaces. The interface body contains the constant and the method declarations for that interface.

Variables declared inside of interface declarations are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Example of interface *Area* and *Shape* are shown below :

```
interface Area
  {
      final static float pi =3.142F;
      float compute (float x, float y);
       void show();
  }
and
interface Shape
   {
        public double area();
        public double volume();
   }
```

5.3.2 Extending Interfaces

Like classes, interfaces can be extended i.e. an interface can be subinterfaced from other interfaces. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface. The syntax for using the extend keyword is shown below :

```
interface name2 extends name1
   {
       body of name2
   }
```

Here, *name2* is the child interface and *name1* is the parent interface.

In following the Sports interface is extended by the Hockey and the Football interfaces.

```
public interface Sports
 {
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
 }

public interface Football extends Sports
 {
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
 }

public interface Hockey extends Sports
 {
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
 }
```

Here, in the Hockey interface has four methods, but it inherits two method from Sports. So, the class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

The ***public*** access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any

number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

5.3.3 Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract class.

To declare a class that implements an interface, you include an **implements** clause in the class declaration as shown in the following :

```
class classname implements Interfacename
        {
            // body of classname
        }
```

For example, we have already define the interface Shape. Let us try to implement it using the Point class.

Program 1 : Implementation of Shape interface

***//* Point.java**

```
interface Shape
  {
        public double area();
        public double volume();
   }
```

```
public class Point implements Shape
 {
    static int x, y;
    public Point()
       {
             x = 0;
             y = 0;
        }

      public double area()
         {
             return 0;
         }
      public double volume()
         {
             return 0;
         }

      public static void print()
         {
             System.out.println("\n Point: " + x + "," + y);
         }

       public static void main(String args[])
         {                          // object declaration
             Point p = new Point();
             p.print();
          }
  }
```

Output :

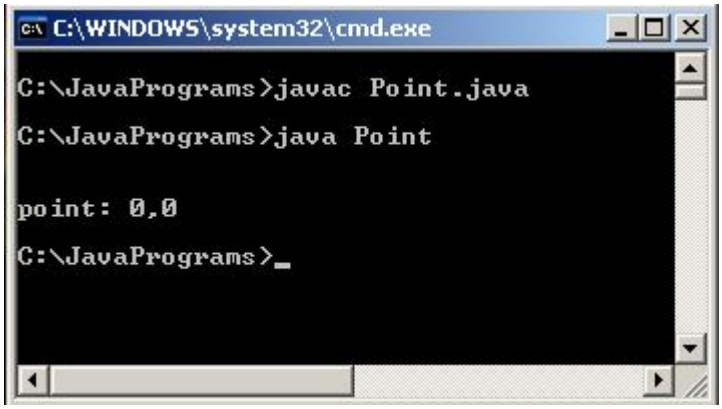


Fig 5.1 Output of Program 1

One class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

5.4 PACKAGES

As we begin to develop more complex programs, we'll start to use more and more classes. Some classes will be from standard libraries; others will be classes which we develop ourselves. When writing a program, often we need to put several subroutine units (e.g. functions) together in a file as a unit, so that the program is manageable. A set of functions as a unit can be loaded and used as needed, and also distributed for other programers to use. This idea of a set of functions as a unit in a file is variously known as ***"library", "package", "module", "add-on".***

In Java, it's called **"package"**.

A Java package is a set of classes and interfaces which are grouped together. The grouping is usually done according to functionality. An example of a package is the JDK Package of Sun Java is shown below :

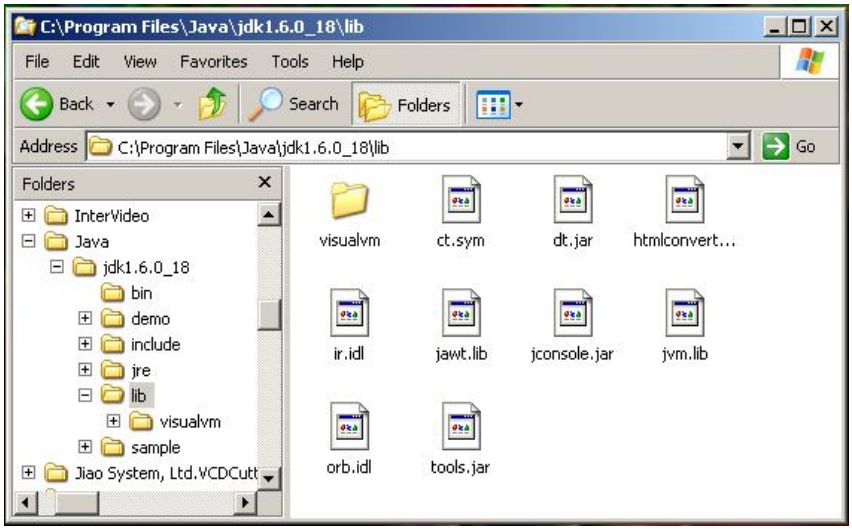


Fig 5.2 JDK Package

5.4.1 Java API Packages

Java packages are classified into two types, they are -

- Java API packages
- User defined packages

Java API provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the Java API. The following figure shows the frequently used packages in programs.

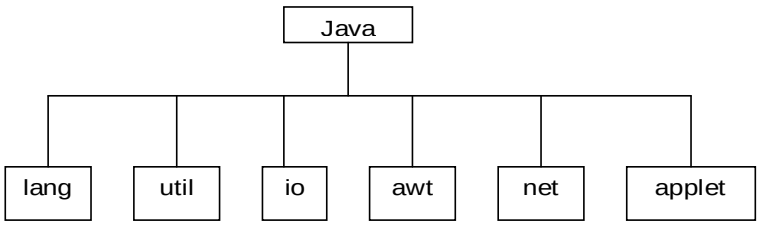


Fig. 5.3 Some API Packages

Java API provides a large number of classes grouped into different packages according to functionality.

java.lang : Language support classes. These are the classes

that Java compiler itself uses and therefore they are automatically imported. They include classes of primitive types, strings, math functions, threads and exceptions.

java.util : Language utility classes such as vector, hash tables, random numbers, data etc.

java.io : Input/output support classes. They provide facilities for the input and output of data

java.awt : set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

java.net : Classes for networking. They include classes for communicating with local computers as well as with internet servers.

java.applet : Classes for creating and implementing applets.

Actually the Java Packages are organised in a hierarchical structure. Java **awt** package is shown in the following fig where it contains the various classes in hierarchical manner.

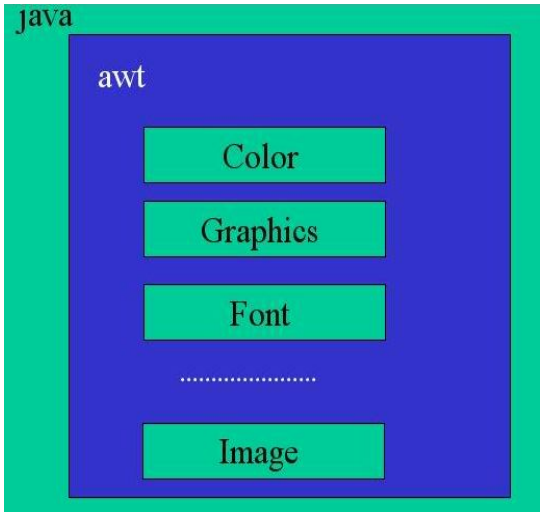


Fig. 5.3 Structure of ***java.awt*** packages

Accessing the classes from packages

The syntax for accessing the package is :

import packagename.classname;
or
import packagename.*;

These are known as *import statement* and should be used at the top of the file, before any class declaration. Its meaning is that we are using the ***classname*** in our program more than one times.

The second statement means for all the classes under the **awt** packages. If we need all the classes under the **awt** package in our program we will use such statement.

For example -

```
import java.awt.Color;      // only import the class color
import java.awt.*;          // import all the class
```

In some situations, suppose we need to access the class only once in our program, then we can make use of it as follows instead of writing import java.awt.Color.

java.awt.Color

5.4.2 Creating User Defined Packages

First declare the name of the package using the ***package*** keyword followed by a package name. This must be the first statement in a java source file.

Declare the package at the beginning of a file using the syntax:

package packagename;

Then define the class which is to be put in the package and declare it as **public**.

Create a subdirectory under the directory where the main source files are stored.

Store the listing as the *classname.java* in the subdirectory created. Compile the file. This creates **.class** file in the subdirectory.

For example, suppose we have a file called ***Test.java***, and we want to put this file in a package say ***DemoPack***. First thing we have to do is to specify the keyword package with the name of the package we want to use (DemoPack in our case) on top of our source file, before the code, that defines the real classes in the package as shown in our Test class below :

```
// only comments can be here

package DemoPack;

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

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierachy of the class. In our case, we have the DemoPack package, which requires only one directory. So, we create a directory DemoPack and put our Test.java into it.

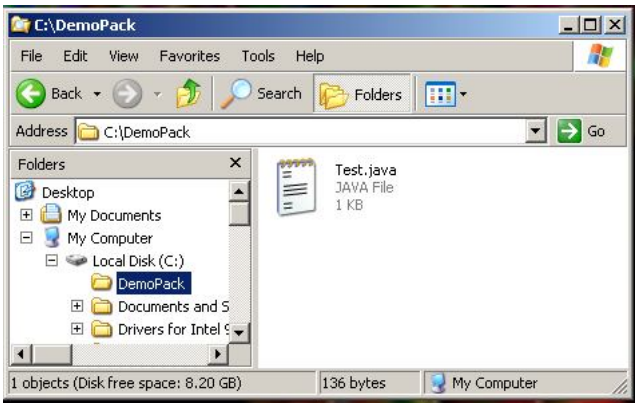


Fig. 5.4 Creating Packages

5.4.3 Accessing and Using Packages

We have already know, how a class can be imported from a package and used in a program. The ***import*** statement is used to search a list of packages for a particular class.

In the case of user defined package also, for accessing a prticular class we will use the import statement. The general form of import statement for searching a class is as follows:

import package1 [.package2] [.package3].classname;

Here *pakcage1* is the name of the top level package, *package2* is the name of the package that is inside the *package1*, and so on.

Now, let us try how to import a user defined package in a program.

We have already created a package named ***DemoPack*** under the DemoPack directory in C dlrive. We will now add an another class to this package by writing and saving the following program in the same directory. While writing you should not forget to add the statement ***package DemoPack;*** at the top of the class.

```
package DemoPack;

public class myClass
  {
    public void displayAll ()
      {
       System.out.println("Demonstration of Packages !!!");
       System.out.println("I Am from myClass !!!");
      }
  }
```

Now, let us write and save the following program in the directory where DemoPack is a subdirectory of it (here in our case

DemoPack is a subdirectory of C drive, so we will save the file in C drive).

```
import DemoPack.myClass;

class  TestClass
   {
        public static void main(String args[])
           {
                                  // new   object created
            myClass obj1 = new myClass();
           obj1.displayAll(); //invokes the method displayAll()
                              }
      }
```

Now, we can compile and run the program which produce the output as shown below :

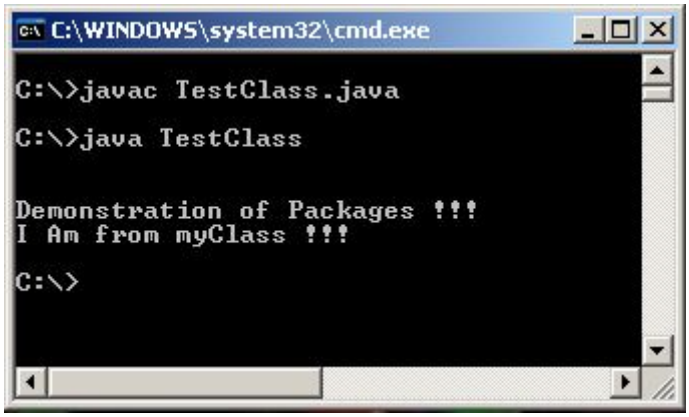


Fig. 5.5 Output of importing Package

Thus, you have learned how to write and import packages.

CHECK YOUR PROGRESS

1. Find the mistake in the following declaration public **interface** DemoInterface

```
{
    void TestMethod(int value)
{
     System.out.println("I am A Simple Interface !");
}
}
```

2. Write the correct form of it.
3. Fill in the blanks of the following :
   (a) ..... support the concept multiple inheritance in Java.
   (b) All the methods of an interface are automatically .....
   (c) ..... clause is used in class declaration for implementing an interface.
   (d) ..... package include the classes of primitive types, strings, math functions, threads and exceptions.
   (e) We will use ..... statement to use all the class under util package.

5.5 LET US SUMUP

1. An interface is a named collection of method definitions (without implementations).
2. An interface can also declare constants.
3. A class can implement one or more interfaces by declaring them in the class definition, and including the code to implement each of the interface methods.
4. A class that does not define all the methods for an interface it implements must be declared as abstract.

5. Classes can be grouped into a package. If a class in a package is to be accessible from outside the package the class must be declared using the keyword *public*.

6. To designate that a class is a member of a package we will use a *package statement* at the beginning of the file containing the class definition.

7. To add classes from a package to a file we will use an import statement immediately following any package statement in the file like *import package1 [.package2] [.package3].classname;*.

5.6 ANSWERS TO CHECK YOUR PROGRESS

1. Inside the body of any interface there should have declaration of methods and variables. Implementation of the methods of interface are done inside the body of the class which implements the interface. Here, in the code, that method should be a declaration but it is given as implementation which is wrong.

2. The correct form of the interface is :

```
public interface DemoInterface
    {
        void TestMethod(int value);
     }
```

3. a) interface b) public c) implement d) java.lang
   e) import java.util.*

5.7 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

5.8 MODEL QUESTIONS

1. What is an interface? Why interface are used ?
2. What is the major difference between an interface and a class?
3. Give an example where interface can be used to support multiple inheritance. Develop a Java program for the example.
4. What are the similarities between interfaces and classes?
5. What is a package? What is its function ?
6. How do we add a class or an interface to a package?
7. How do we design a package?
8. Discuss the Java API Packages.

******

UNIT-6 EXCEPTION HANDLING

UNIT STRUCTURE

6.1 LEARNING OBJECTIVES

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

- learn the concept of exceptions in Java
- learn to use the keywords *throw*, *try*, *catch*, and *finally* in exception handling
- learn about the *Throwable* class hierarchy
- learn about unchecked and checked exception in Java
- throw exceptions implicitly as well as explicitly
- learn about how to catch exceptions
- learn to handle user-defined exceptions and will be able to create your own exception

6.2 INTRODUCTION

By now, you must have been acquainted with so many concepts of Java programming language such as *data types and variables, operators, control flow statements etc.* We have also presented the concept of classes, objects, methods, constructors, inheritance, various types of modifiers, arrays, strings, vectors as well as the interfaces and packages. These concepts will help the learners to write and develop suitable programs.
While writing program there may arise some errors for some mistakes. An error may produce an incorrect output or may terminate the execution of the program abruptly. It is therefore important to detect and manage those errors. Java facilitates the management of such situation by handling exceptions.

In this unit, we will discuss how exceptions can be handled in Java programming language. The unit describes when and how to use exceptions.

6.3 ERRORS AND EXCEPTIONS

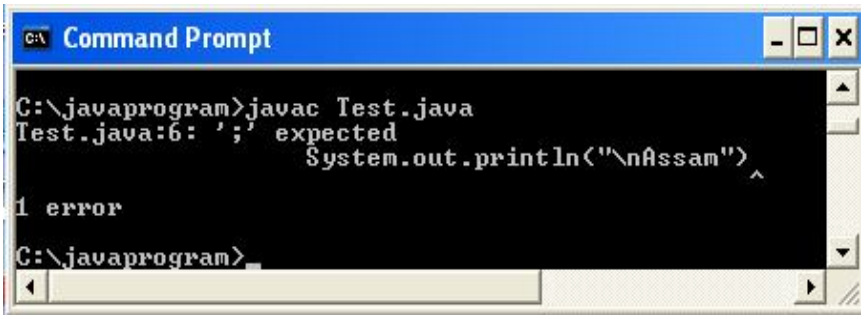
We can define an ***exception*** as an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. The Java programming language uses *exceptions* to handle errors and other exceptional events. Exceptions are used in a program to signal that some error or exceptional situation has occurred, and that it does not make sense to continue the program flow until the exception has been handled.

There are two categories of errors: ***Compile-time errors*** and ***Run-time errors***. All syntax errors detected and displayed by the Java compiler are termed as ***Compile-time errors***. If the compiler detects an error while compiling a program, then the ***.class*** file will not be created. For successful compilation we have to correct the syntax error first. Let us consider the following example for the demostration of compile time error:

Program 6.1: Test.java (Program showing Compile-time error)

```
class Test
{
        public static void main(String args[ ])
       {
                System.out.println("KKHSOU");
                System.out.println("\nAssam")
        }
}
```

While compiling the above program ***Test.java***, the following message will be displayed on the screen:



We can see that the statement

System.out.println("\nAssam")

has no semicolon at the end. The Java compiler displays where the errors are in the program. We can then correct the errors in the appropriate line and recompile the program. If there is no other error in the program then it will create ***.class*** file (here, *Test.java*). We can then run the program to see the output. Some of the most common compile-time errors are:

- Use of variable without declaration
- Missing brackets in classes and methods
- Missing semicolons
- Incompatible assignment statements etc.

Sometimes, a program may compile successfully creating ***.class*** file but may not execute properly i.e., they may produce wrong result or may terminate abruptly. These errors may occur due to wrong logic of the program and many more reasons like

- Dividing an integer by zero
- Converting invalid strings to number
- Trying to store a value into an array of incompatible class or type etc.

Such types of errors are termed as ***Run-time errors***. Let us consider the following example for the demonstration of run-time error:

//Program 6.2: Error.java (Demonstration of Run-time error and
// Implicitly throwing an exception)

```
class Error
{
    public static void main(String args[])
    {
        int p, q, r, result;
        p = 20;
        q = 6;
        r = 6;
        result = p / (q-r);    //(q-r) is equal to zero
        System.out.println("The result is : " + result);
    }
}
```

When we compile the above program, then it will create the ***Error.class*** file as the compilation is successful. The compiler will not display any error message as there is no syntax error in the code. When we try to run the program then it will display the following error message and terminate the execution of the program. In the statement **Result = p/(q-r);** we are dividing p by zero. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred)

An **exception** in Java is an object that is created when an abnormal situation arises in a program. When an error occurs within a method, the method creates an object and hands it off to the runtime system. This exception object has data members that store information about the nature of the problem. Such an object can be instantiated by a running program in two ways:

> ***explicitly*** by a ***throw*** statement in the program
> or ***implicitly*** by the Java run-time system when it is unable to execute a statement in a program*(*as in *Program 6.2)*.

One major benefit of having an error signaled by an exception is that it separates the code that deals with errors from the code that is executed when things are moving along smoothly. Another positive aspect of exceptions is that they provide a way of enforcing a response to particular errors – with many kinds of exceptions, we must include code in our program to deal with them, otherwise our code will not compile. One important idea to grasp is that not all errors in our programs need to be signaled by exceptions.

At this point we will discuss the basics of Java's exception throwing and catching mechanism. When an error occurs in a Java program it usually results in an exception being thrown. A method may throw an exception for many reasons, for instance if the input parameters are invalid (negative when expecting positive etc.). How we ***throw***, ***catch*** and ***handle*** these exception matters. There are several different ways to do so.

Creating an exception object and handing it to the runtime system is called ***throwing an exception***. When an exception is thrown, it can be caught by a ***catch*** clause of ***try*** statement. If the exception object is not caught and handled properly, the interpreter will display an error message (e.g., like the output of *Program 6.2*) and terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and display an appropriate message. This process is known as ***exception handling***.

6.4 EXCEPTIONS HIERARCHY

The hierarchy of exception classes commence from ***Throwable*** class which is the base class for an entire family of exception classes, declared in

java.lang package as **java.lang.Throwable**. An exception is always an object of some subclass of the standard class ***Throwable***. Two direct subclasses of the Throwable class are the ***Error*** class and the ***Exception*** class which cover all the standard exceptions. Both these classes themselves have subclasses which identify specific exception conditions. The following figure 6.1 shows the exception hierarchy which gives some of the standard exception in Java.

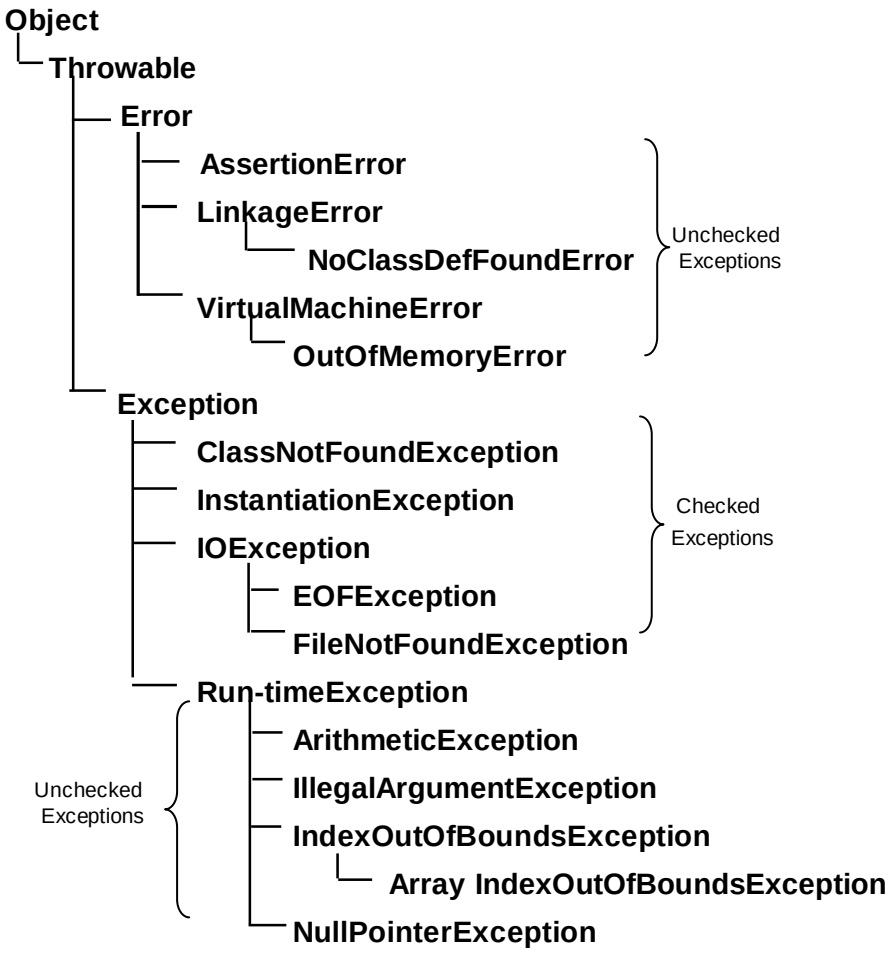


Fig. 6.1: Exception Hierarchy in Java

6.4.1 Checked versus Unchecked Exceptions

In Java there are basically two types of built-in exceptions: ***Unchecked exceptions*** and ***checked exceptions***. Exception in Java are classified on the basis of the exception handled by the java compiler.

- **Unchecked Exceptions**

The kinds of exception that can be prevented by writing better code are *unchecked exceptions*. They are instances of the ***Error*** class, the ***Run-timeException*** class, and their extensions. These exception arises during run-time ,that occur due to invalid argument passed to method.

- **Checked Exceptions**

The ***checked exceptions*** are checked by the compiler before the program is run. At compile time, the Java compiler checks that a program contains handlers for checked exceptions. These exception are the object of the ***Exception*** class or any of its subclasses except *Run-timeException* class. These condition arises due to invalid input, problem with our network connectivity and problem in database. The statements that throw them either must be placed within ***try*** statement or they must be declared in their ***method's header***.

The **java.lang** package defines several classes and exceptions. Some of these classes are not checked while some other classes are checked (Table 6.1).

EXCEPTIONS	DESCRIPTION	CHECKED	UNCHECKED
ArithmeticException	Arithmetic errors such as a divide by zero	-	YES
ArrayIndexOutOfBoundsException	Arrays index is not within array.length	-	YES
ClassNotFoundException	Related Class not found	YES	-
IOException	InputOuput field not found	YES	-
IllegalArgumentException	Illegal argument when calling a method	-	YES
InterruptedException	One thread has been interrupted by another thread	YES	-
NoSuchMethodException	Nonexistent method	YES	-
NullPointerException	Invalid use of null reference	-	YES
NumberFormatException	Invalid string for conversion to number	-	YES

Table 6.1

6.5 HANDLING EXCEPTIONS

If we want to deal with the exceptions where they occur, there are three kinds of code block that we can include in a method to handle them. These are ***try***, ***catch***, and ***finally***. At this point we will first discuss the detail of *try* and *catch* blocks and will come to the application of a *finally* block a little later.

- **The try Block**

When we want to catch an exception, the code in the method that might cause the exception to be thrown must be enclosed in a **try** block. A **try** block is simply the keyword *try*, followed by braces enclosing the code that can throw the exception:

```
try
{
        // Code that can throw one or more exceptions
}
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block. It should be remebered that every ***try*** statement should be followed by at least one ***catch*** statement if there is no *finally* block.

- **The catch Block**

We enclose the code to handle an exception of a given type in a ***catch*** block. The catch block must immediately follow the try block that contains the code that may throw that particular exception. A *catch* block consists of the keyword *catch* followed by a parameter between parentheses that identifies the type of exception that the block is to deal with. This is followed by the code to handle the exception enclosed between braces:

```
try
{
      // Code that can throw one or more exceptions
}
catch( ArithmeticException e)
{
      // Code to handle the exception
}
```

The above catch block only handles *ArithmeticException* exceptions. This implies that, this is the only kind of exception that can be thrown in the try block. If others can be thrown, this will not compile. Let us modify *Program 6.2* for the demonstration of try and catch blocks to handle an arithmetic expression.

//Program 6.3: Demo.java

```
class Demo
{
   public static void main(String args[])
   {
      int p, q, r, x, y;
      p = 20;
      q = 5;
      r = 5;
      try
      {
          x = p / (q-r);    //exception here
      }
      catch(ArithmeticException e)
      {
         System.out.println("Testing an exception");
      }
      y = p / (q+r);
      System.out.println("\nThe result is = "+y);
   }
}
```

If we run the above program, the following output will be displayed

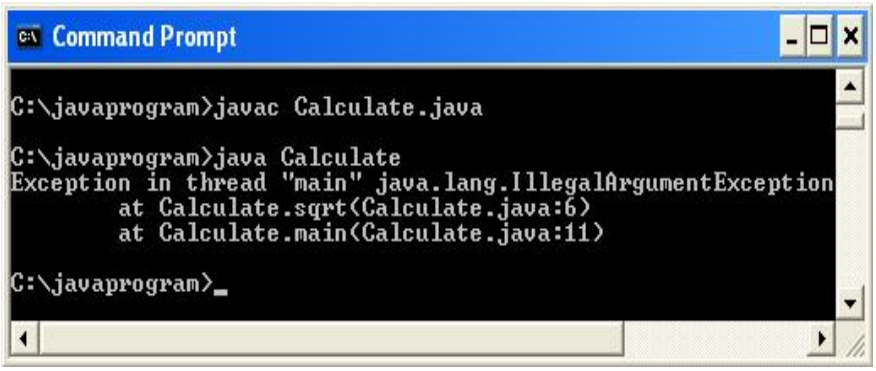
The execution of the program does not stop at the point of exceptional condition. It catches the error condition, displays the message “Testing an exception”. The execution continues and gives the result without terminating the program as if nothing has happened.

Program 6.3 is an example of implicitly throwing and catching an unchecked exception. Let us consider the following program for an illustration of unchecked exception that is thrown by an *explicit* throw statement.

//Program 6.4: Calculate.java (Explict throw of unchecked exception)

```
class Calculate
{
        static double sqrt(double n)
        {
                if(n<0)
                        throw new IllegalArgumentException();
                return Math.sqrt(n);
        }
       public static void main(String[ ] args)
       {
                System.out.println(sqrt(-16));
                System.out.println("\nEnd of Calculate Method");
       }
}
```

When we run the program after compiling, then it will display the following and terminate the program.



The output is showing what happens when an exception is not caught.
It can be prevented with the help of try and catch statement. The following program is a modification of Program 6.4 to handle such unchecked exception.

//***Program 6.5***: Calculate.java (Catching an unchecked exception
//which is thrown explicitly)

```
class Calculate
{
    static double sqrt(double n)
    {
       if(n<0)
            throw new IllegalArgumentException();
       return Math.sqrt(n);
     }
    public static void main(String[ ] args)
    {
           try
          {
                System.out.println(sqrt(-16));
           }
           catch(Exception exception)
           {
                System.out.println("exception: "+ exception);
           }
          System.out.println("\nThe exception was caught");
          System.out.println("\nEnd of Calculate Method");
    }
}
```

The output will be like this:

The ***IllegalArgumentException*** object is thrown with the statement **throw new IllegalArgumentException();** and that exception is caught with the statement **catch(Exception exception)** as it is generated by the statement **System.out.println(sqrt(-16));** within that try block. The exception in the Program 6.5 can be handled by writing proper program code.

Normally, the try statement should be used only for checked exceptions. That is because the purpose of try statement is to handle unanticipated errors. Exceptions should be reserved for the unusual or catastrophic situations that can arise. The reason for this is that dealing with exceptions involves quite a lot of processing overhead, so if our program is handling exceptions a lot of the time it will be a lot slower than it needs to be.

Following are the list of various checked exception that defined in the **java. lang** package.

Exception	**Reason for Exception**
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program
Instantiation Exception	This Exception occurs when you create an object of an abstract class and interface

Illegal Access Exception	This Exception occurs when you create an object of an abstract class and interface.
Not Such Method Exception	This Exception occurs when the method you call does not exist in class

6.5.1 Multiple Catch Blocks

If a ***try*** block can throw several different kinds of exception, we can put several ***catch*** blocks after the try block to handle them.

```
try
{
        // Code that may throw exceptions
}
catch(ArithmeticException e)
{
        // Code for handling ArithmeticException exceptions
}
catch(IndexOutOfBoundsException e)
{
    // Code for handling IndexOutOfBoundsException exceptions
}
    // Execution continues here...
```

Exceptions of type ***ArithmeticException*** will be caught by the first catch block, and exceptions of type ***IndexOutOfBoundsException*** will be caught by the second. Of course, if an ArithmeticException exception is thrown, only the code in that catch block will be executed.When it is complete, execution continues with the statement following the last catch block.When we need to catch exceptions of several different types for a try block, the order of the catch blocks is important. When an exception is thrown, it will

be caught by the first catch block that has a parameter type that is the same as that of the exception, or a type that is a superclass of the type of the exception.

An extreme case would be if we specify the catch block parameter as type ***Exception***. This will catch any exception that is of type *Exception*, or of a class type that is derived from *Exception*. This includes virtually all the exceptions we are likely to meet in the normal course of events.This has implications for multiple catch blocks relating to exception class types in a hierarchy. The catch blocks must be in sequence with the most ***derived type first***, and the ***most basic type last***. Otherwise our code will not compile. The simple reason for this is that if a catch block for a given class type precedes a catch block for a type that is derived from the first, the second catch block can never be executed and the compiler will detect that this is the case

Suppose we have a catch block for exceptions of type *ArithmeticException*, and another for exceptions of type *Exception* . If we write them in the following sequence, exceptions of type *ArithmeticException* could never reach the second catch block as they will always be caught by the first.

```
// Invalid catch block sequence – will not compile!
try
{
        // try block code
}
catch(Exception e)
{
        // Generic handling of exceptions
}
catch(ArithmeticException e)
{
        // Specialized handling for these exceptions
}
```

The above code will not compile.Thus if we have catch blocks for several exception types in the same class hierarchy, we must put the catch blocks

in order, starting with the lowest subclass first, and then progressing to the highest superclass.In principle, if you are only interested in generic exceptions, all the error handling code can be localized in one catch block for exceptions of the superclass type. However, in general it is more useful, and better practice, to have a catch block for each of the specific types of exceptions that a try block can throw.

6.5.2 The finally Block

The immediate nature of an exception being thrown means that execution of the try block code breaks off, regardless of the importance of the code that follows the point at which the exception was thrown. This introduces the possibility that the exception leaves things in an unsatisfactory state. We might have opened a file, for instance, and because an exception was thrown, the code to close the file is not executed.

The ***finally*** block provides the means to clean up at the end of executing a try block. We use a finally block when we need to be sure that some particular code is run before a method returns, no matter what exceptions are thrown within the previous try block. A finally block is always executed, regardless of what happens during the execution of the method. If a file needs to be closed, or a critical resource released, we can guarantee that it will be done if the code to do it is put in a *finally* block. The finally block has a very simple structure:

```
finally
{
        // Clean-up code to be executed last
}
```

Just like a catch block, a finally block is associated with a particular try block, and it must be located immediately following any catch blocks for the try block. If there are no catch blocks then we position the finally block immediately after the try block. If we do not do this, our program will not compile. Java provides the ***finally*** statement that can be used handle an exception that is not caught by any of the previous catch statement. i.e., a try statement does not have to have a catch block if it has a ***finally*** block. If

the code in the try statement has multiple exit points and no associated catch clauses, the code in the finally block is executed no matter how the try block is exited. Thus, it makes sense to provide a finally block whenever there is code that must always be executed.

6.6 USER- DEFINED EXCEPTIONS

As we come across *Built-in* exception, we create own customized exception as per requirements of the application. On each application there is a specific constraints. Error-handling become necessary while developing a constraint application. For example, suppose in the case of a banking application, a customer whose age is less than 18 need to open Joint Account. The Exception class and its subclass in Java is not able to meet up the required constraint in application. For this, we create our own customized exception to over address these constraints and ensure the integrity in the application. Let us see how to handle and create *user-defined* exception. The keywords try, catch and finally are used in implementing user-defined exceptions. This ***Exception*** class inherits all the method from Throwable class.

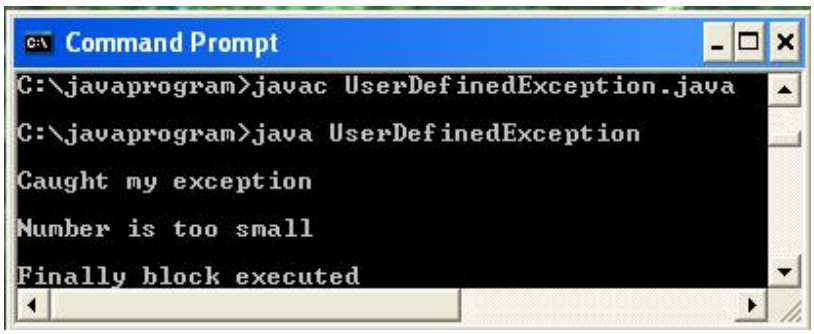
In the following program, a class ***MyException*** is created which is a subclass of the ***Exception*** class. The ***MyException*** class has one constructor,i.e ***MyException***( ).

```
//Program 6.7: UserDefinedException.java
import java.lang.Exception;
class MyException extends Exception
{
      MyException(String m)
      {
            super(m);
      }
}
class UserDefinedException
{
   public static void main(String args[ ])
   {
```

```
        int a=5, b=5000;
        try
        {
              float c =(float)a/(float)b;
              if(c<0.01)
              {
                    throw new MyException("\nNumber is too small");
              }
        }
        catch(MyException e)
        {
              System.out.println("\nCaught my exception");
               System.out.println(e.getMessage());
        }
        finally
        {
               System.out.println("\nFinally block executed");
        }
   }
}
```

The object **e** which contain the error message "*Number is too small*" is caught by the catch block which then displays the message using the ***getMessage()*** method. The output will be like this:

We can also learn how to use the statement ***finally*** with the above program. The last line of the output is produced by the ***finally*** block.



CHECK YOUR PROGRESS

1. Is the following code legal?

```
try
{
        ......
        ......
 }
  finally {
        .......
  }
```

2. What exception types can be caught by the following handler?

```
catch (Exception e) {
}
```

What is wrong with using this type ofexception handler?

3. Is there anything wrong with the following exception handler as written? Will this code compile?

```
try
{
     .......
}
catch (Exception e)
{
 }
 catch (ArithmeticException a)
{
 }
```

4. How many catch clauses can a try statement have?
5. What happens if a thrown exception is not caught?

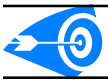
6.8 LET US SUM UP

The important concepts we have explored in this unit are:

- *Exception* that means exceptional errors. Actually *exceptions* are used

for handling errors in programs that occurs during the program execution.

- Error that occurs during the program execution generate a specific object which has the information about the errors occurred in the program. Exceptions are objects of subclasses of the class Throwable.
- Java includes a set of standard exceptions that may be thrown automatically, as a result of errors in our code, or may be thrown by methods in the standard classes in Java.
- If a method throws exceptions that are not caught, and are not represented by subclasses of the class Error, or by subclasses of the class RuntimeException, then we must identify the exception classes in a throws clause in the method definition.
- If we want to handle an exception in a method, we must place the code that may generate the exception in a try block. A method may have several try blocks.
- Exception handling code is placed in a catch block that immediately follows the try block that contains the code that can throw the exception. A try block can have multiple catch blocks that deal with different types of exception.
- A finally block is used to contain code that must be executed after the execution of a try block, regardless of how the try block execution ends. A finally block will always be executed before execution of the method ends.
- We can define our own exception classes that, in general, should be derived from the class Exception.



6.9 ANSWERS TO CHECK YOUR PROGRESS

Answer 1 : Yes, it is legal and very useful. A try statement does not have to have a catch block if it has a finally block. If the code in the try statement has multiple exit points and no associated catch clauses, the code in the finally block is executed no matter how the try block is exited. Thus it makes sense to provide a finally block whenever there is code that must always be executed. This include resource recovery code, such as the code to close I/O streams.

Answer 2 : This handler catches exceptions of type Exception; therefore, it

catches any exception. This can be a poor implementation because we are losing valuable information about the type of exception being thrown and making our code less efficient. As a result, our program may be forced to determine the type of exception before it can decide on the best recovery strategy.

Answer 3 : This first handler catches exceptions of type Exception; therefore, it catches any exception, including ArithmeticException. The second handler could never be reached. This code will not compile.

Answer 4: A try statement may have any number of catch clauses, as long as it has atleast one or a finally clause. Each catch clause must catch a different type of exception.

Answer5: If a thrown exception is not caught, then the program will terminate immediately.

6.10 FURTHER READINGS

1. "*Java Programming Language Handbook* ", by Anthony Potts, David H. Friedel Jr. , Coriolis Group Books

2. "*JAVA How to Program*", Deitel & Deital, PHI Publication

3. "*Programming with Java-A Primer*" by E Balagurusamy, Tata McGrawHill

6.11 POSSIBLE QUESTIONS

1. What is an exception?
2. Is it essential to catch all types of exception?
3. How many catch blocks can we use with one try block?
4. What is a finally block? When and how is it used? Give a suitable example.
5. How do we define a try block?
6. How do we define a catch block?
7. List some of the common types of exceptions that might occur in Java. Give examples.
8. What do you mean by checked and unchecked exceptions? Give examples.

*****

UNIT - 7 FILE HANDLING

UNIT STRUCTURE

7.1 LEARNING OBJECTIVES

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

- learn about the most important io package in Java
- learn about the predefined streams
- learn about console input and output
- describe how to read data from file and how to write data to file

7.2 INTRODUCTION

In this unit we will learn one of the Java's most important package **io**. We will learn about the streams, different stream classes. Besides this how to read data from input and how to write data into output. We also give a brief introduction of file handling in Java.

The io package supports Java's basic I/O (Input/Output) system including file I/O. Java program perform I/O through streams. A stream is linked to a physical device by the Java I/O system. Java define 2 types of stream, *byte*

and *character*. In Java 1.0, the only way to perform console input was to use a byte stream. The preferred method of reading console input for Java 2 is use a character oriented stream, which makes the program easier to internationalize and maintain.

Java provides a number of classes and methods that allow to read and write files. In Java all files are byte oriented, and Java provides method to read and write bytes from and to a file.

7.3 I/O BASICS : STREAMS

Java views each file as a sequential stream of bytes. Each operating system provides a mechanism to determine the end of a file, such as an end-of-file marker or count of the total bytes in the file that is recorded in a system-maintained administrative data structure.

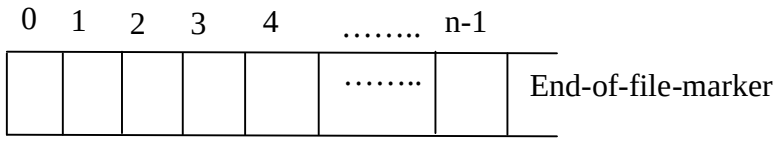
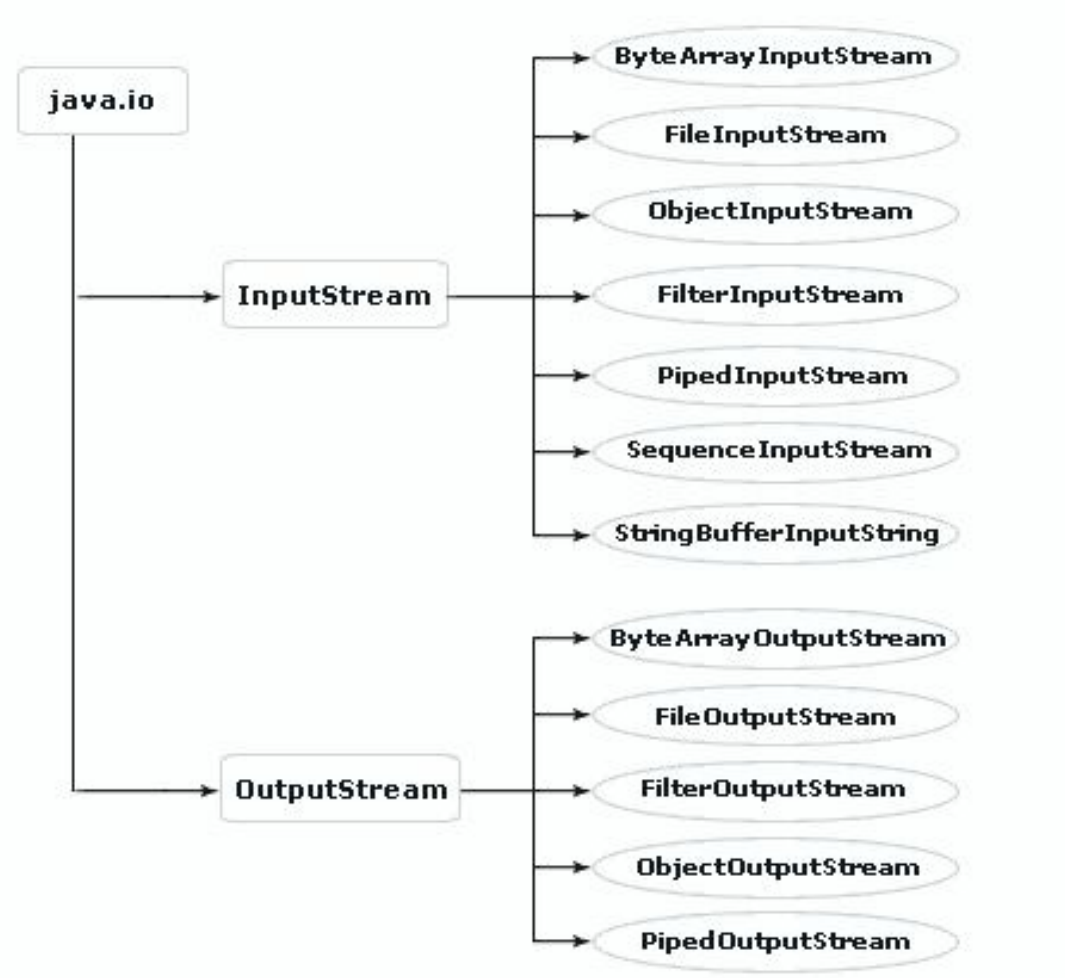


Fig: Java's view of a

A Java program processing a stream of bytes simply receives an indication from the operating system when the program reaches the end of the stream-the program does not need to know how the underlying platform represents files or streams.

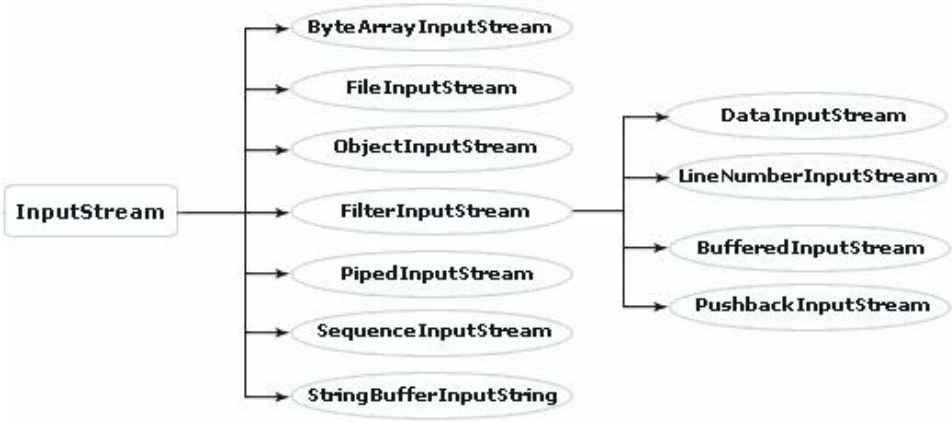
The Java ***Input/Output (I/O)*** is a part of ***java.io*** package. The java.io package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources. The InputStream and OutputStream are central classes in the package which are used for reading from and writing to byte streams, respectively.

The ***java.io*** package can be categorised along with its stream classes in a hierarchy structure as shown below:



The ***InputStream*** class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a file, a string, or memory that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when we create it. We can explicitly close a stream with the *close( )* method, or let it be closed implicitly when the object is found as a garbage.

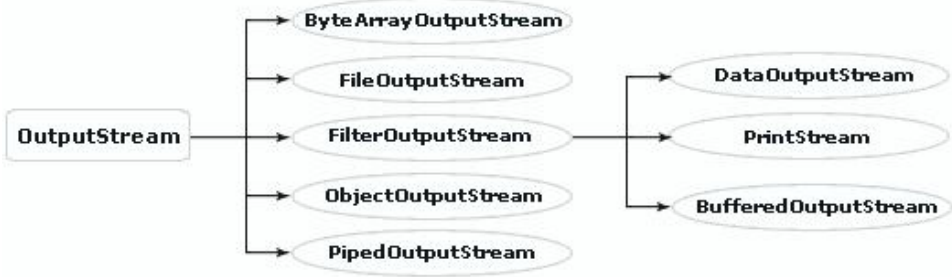
The subclasses inherited from the ***InputStream*** class can be seen in a hierarchy manner as shown below:



InputStream is inherited from the ***Object*** class. Each class of the InputStreams provided by the java.io package is intended for a different purpose.

The ***OutputStream*** class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the close() method, or let it be closed implicitly when the object is garbage collected.

The classes inherited from the OutputStream class can be seen in a hierarchy structure shown below:

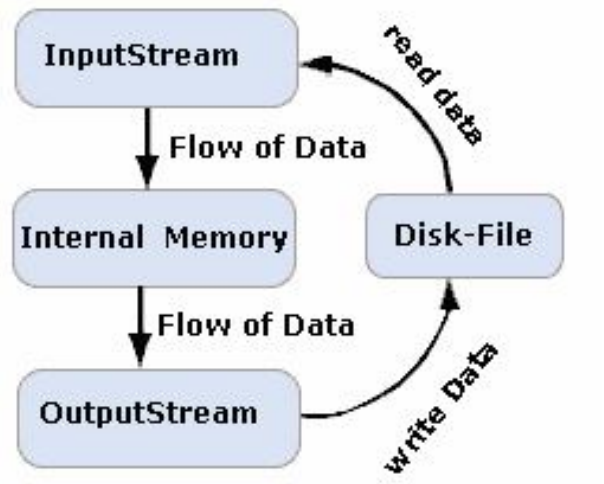


OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the java.io package is intended for a different purpose.

How Files and Streams Work:

Java uses streams to handle I/O operations through which the data is flowed from one location to another. For example, an InputStream can flow the data from a disk file to the internal memory and an OutputStream can flow the data from the internal memory to a disk file. The disk-file may be a text file or a binary file. When we work with a text file, we use a character stream where one character is treated as per byte on disk. When we work with a binary file, we use a binary stream.

The working process of the I/O streams can be shown in the given diagram.



7.4 THE STREAM CLASSES

There are two types of streams

1. Byte –for Binary I/O
2. Character – for Character I/O

Programs use byte streams to perform input and output of 8-bit bytes.

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way, they differ mainly in the way they are constructed.

We'll explore FileInputStream and FileOutputStream by examining an example program named CopyBytes.java, which uses byte streams to copy kkhsou.txt, one byte at a time and write the content of the file on outagain.txt.

Kkhsou.java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
   public static void main(String[] args) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;
       try {
          in = new FileInputStream("kkhsou.txt");
          out = new FileOutputStream("outagain.txt");
           int c;
           while ((c = in.read()) != -1) {
              out.write(c);
           }

      } finally {
          if (in != null) {
              in.close();
          }
          if (out != null) {
              out.close();
          }
      }
   }
}
```

We need to create one file kkhsou.txt which contains the test "DISHPUR GUWAHATI". The program will copy this text into a new file called outagain.txt.

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.

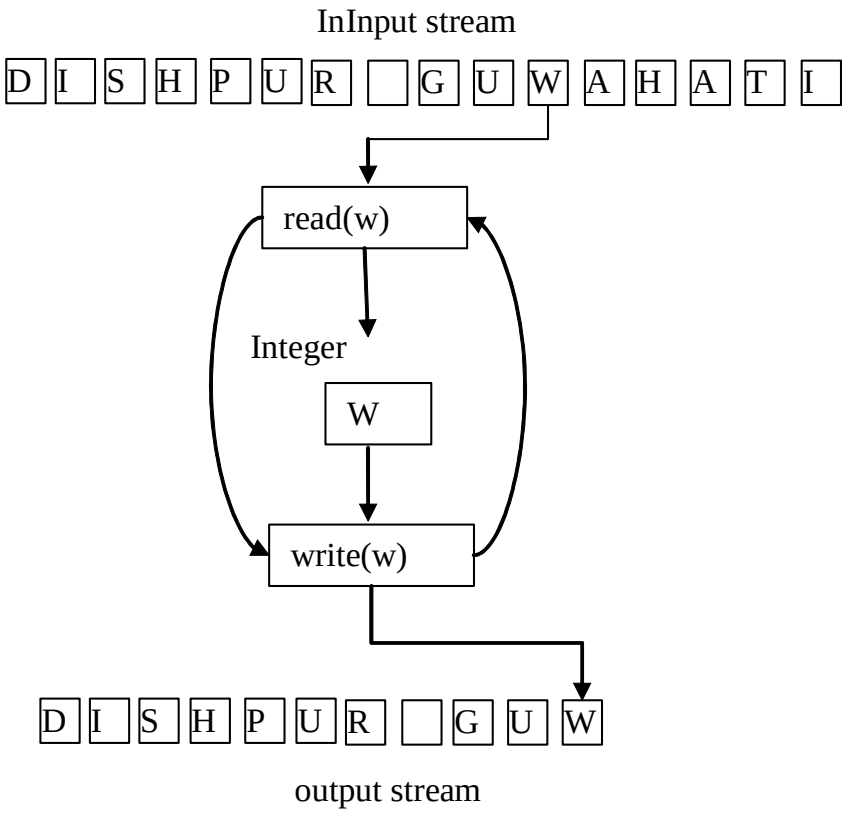


Fig: Simple byte stream input and output.

We can notice that read() returns an int value. Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.

The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize.

Java represents strings in ***Unicode***, an International standard character encoding that is capable of representing most of the world's written languages. Typical human-readable text files, however, use encodings that are not necessarily related to Unicode, or even to ASCII, and there are many such encodings. Character streams hide the complexity of dealing with these encodings by providing two classes that serve as bridges between byte streams and character streams. The InputStreamReader class implements a character-input stream that reads bytes from a byte-input stream and converts them to characters according to a specified encoding.

Similarly, the OutputStreamWriter class implements a character-output stream that converts characters into bytes according a specified encoding and writes them to a byte-output stream.

A second advantage of character streams is that they are potentially much more efficient than byte streams. The implementations of many of Java's original byte streams are oriented around byte-at-a-time read and write operations. The character-stream classes, in contrast, are oriented around buffer-at-a-time read and write operations. This difference, in combination with a more efficient locking scheme, allows the character stream classes to more than make up for the added overhead of encoding conversion in many cases.

7.5 THE PREDEFINED STREAMS

All Java programs automatically import **java.lang** package. This package defines a class called System, which encapsulates several aspects of the run-time environment. For example, using some of its method we can obtain the current time and the settings of various properties associated within the system. System also contains three predefined system variables, ***in***, ***out*** and ***err***.

System.out refers to the standard output stream, which is console by default. System.in refers to standard input, which is the keyboard by default. System.err refers to the standard error stream, which is the console.

System.in is an object of type ***InputStream***, ***System.out*** and ***System.err*** are object of type ***PrintStream***. These are byte stream.

CHECK YOUR PROGRESS -1

1. Java views each file as a sequential ..... of bytes.
2. The Java Input/Output (I/O) is a part of ..... package.
3. Java uses ..... to handle I/O operations through which the data is flowed from one location to another
4. Java represents strings in .....
5. java.lang package defines a class called.....
6. State whether the following statements are true or false:
   a. Each operating system provides a mechanism to determine the end of a file, such as an end-of-file marker or count of the total bytes in the file Applet are not supported by web browser.
   b. The InputStream class is used for reading the data such as a byte and array of bytes from an input source.
   c. There are three types of streams: byte, character, float

7.6 READING CONSOLE INPUT

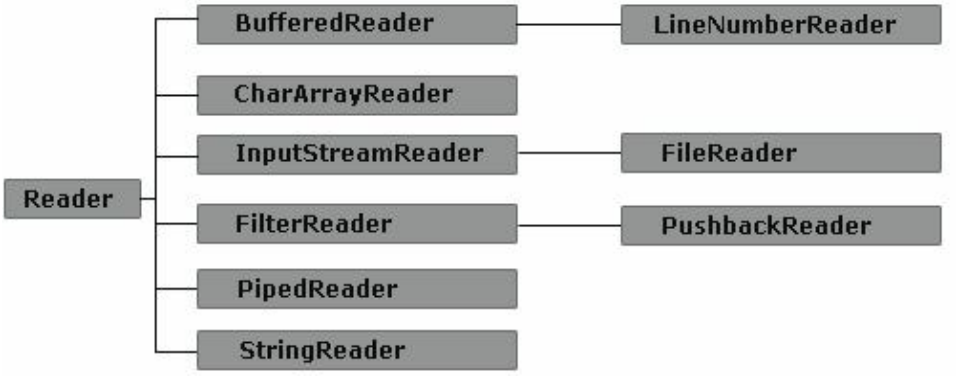
Java also supports three Standard Streams:

- **Standard Input**: Accessed through System.in which is used to read input from the keyboard.
- **Standard Output**: Accessed through System.out which is used to write output to be display.
- **Standard Error**: Accessed through System.err which is used to write error output to be display.

Working with Reader classes:

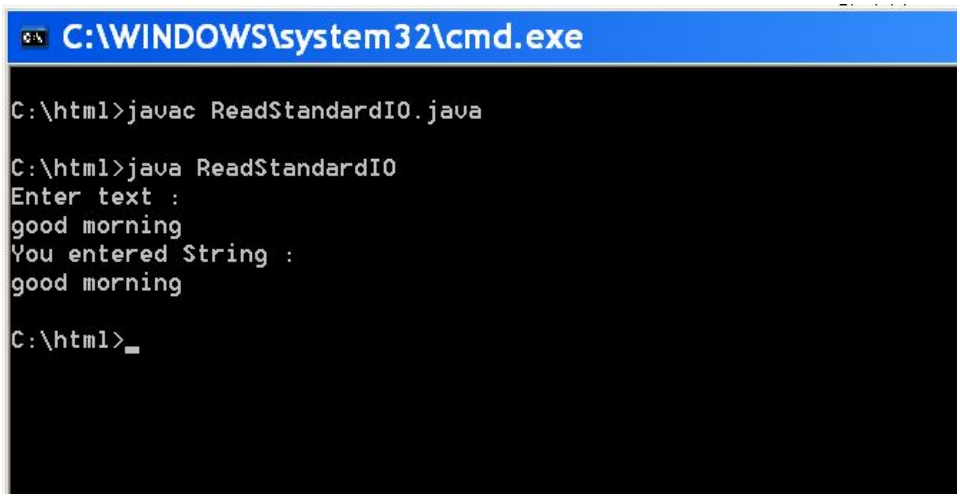
Java provides the standard I/O facilities for reading text from either the file or the keyboard on the command line. The Reader class is used for this purpose that is available in the java.io package. It acts as an abstract class for reading character streams. The only methods that a subclass must implement are read(char[ ], int, int) and close(). the Reader class is further categorized into the subclasses.

The following diagram shows a class-hierarchy of the java.io.Reader



This program illustrates you how to use standard input stream to read the user input.

```
import java.io.*;
public class ReadStandardIO{
public static void main(String[] args) throws IOException{
                InputStreamReader inp=new
        InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(inp);
                System.out.println("Enter text : ");
                String str = br.readLine();
                System.out.println("You entered String : ");
                System.out.println(str);
        }
}
```

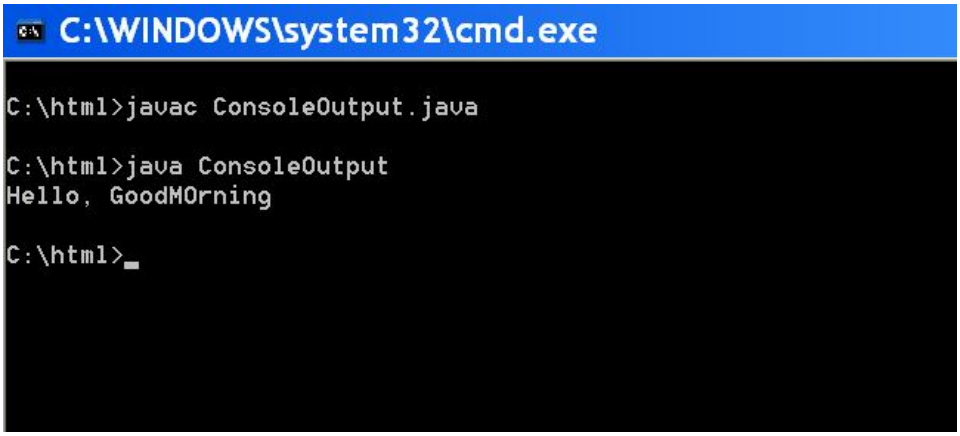


7.7 WRITING CONSOLE OUTPUT

We can write programs that write text lines to the “console”, which is typically a DOS command window.

ConsoleOutput.java:

```
public class ConsoleOutput {
   public static void main(String[] args) {
      System.out.println(“Hello, GoodMOrning”);
   }
}
```



No imports are required, The System class is automatically imported (as are all java.lang classes).

You can write one complete output line to the console by calling the System.out.println() method. The argument to this method will be printed. println comes from Pascal and is short for “print line”. There is also a similar print method which writes output to the console, but doesn’t start a new line after the output.

7.8 READING AND WRITING FILES

Java provides a number of classes and methods that allow us to read and write files. In Java all files are byte oriented, and Java provides methods to read and write bytes from and to a file.

The **File** class deals with the machine dependent files in a machine-independent manner i.e., it is easier to write platform-independent code that examines and manipulates files using the File class. This class is available in the java.lang package.

The **java.io.File** is the central class that works with files and directories. The instance of this class represents the name of a file or directory on the host file system.

When a File object is created, the system does not check to the existence of a corresponding file/directory. If the file exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, reading from or writing to it.

Lets understand some I/O streams that are used to perform reading and writing operation in a file.

Java supports the following I/O file streams.

- FileInputStream
- FileOutputStream
- FileInputstream

This class is a subclass of Inputstream class that reads bytes from a specified file name . The read() method of this class reads a byte or array of bytes from the file. It returns -1 when the end-of-file has been reached. We typically use this class in conjunction with a BufferedInputStream and DataInputstream class to read binary data. To read text data, this class is used with an InputStreamReader and BufferedReader class. This class throws FileNotFoundException, if the specified file is not exist. We can use the constructor of this stream as:

FileInputstream(File filename);

FileOutputStream:

This class is a subclass of OutputStream that writes data to a specified file name. The write() method of this class writes a byte or array of bytes to the file. We typically use this class in conjunction with a BufferedOutputStream and a DataOutputStream class to write binary data. To write text, we typically

use it with a PrintWriter, BufferedWriter and an OutputStreamWriter class. You can use the constructor of this stream as:

FileOutputstream(File filename);

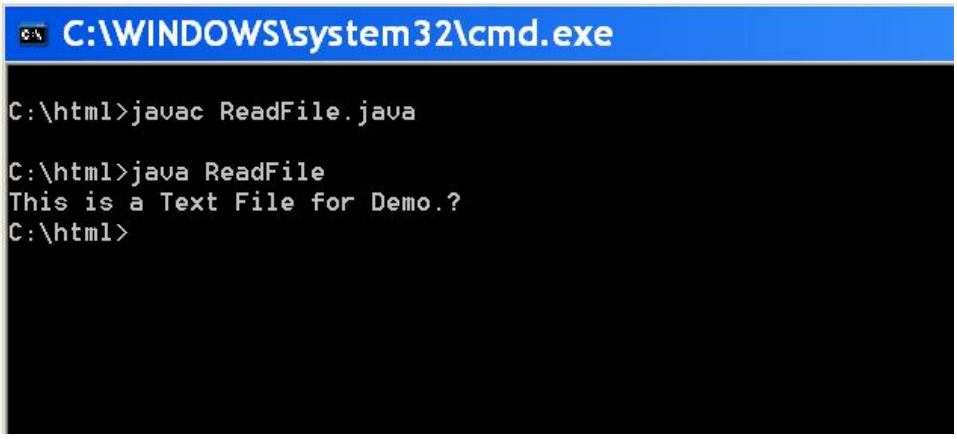
DataInputStream:

This class is a type of FilterInputStream that allows you to read binary data of Java primitive data types in a portable way. In other words, the DataInputStream class is used to read binary Java primitive data types in a machine-independent way. An application uses a DataOutputStream to write data that can later be read by a DataInputStream. You can use the constructor of this stream as:

DataInputStream(FileOutputstream finp);

The following program demonstrate, how the contains are read from a file.

```
import java.io.*;

public class ReadFile{
   public static void main(String[] args) throws IOException{
     File f;
   f=new File("demo.txt");
     if(!f.exists()&& f.length()<0)
     System.out.println("The specified file is not exist");

      else{
        FileInputStream finp=new FileInputStream(f);
      byte b;
    do{
     b=(byte)finp.read();
     System.out.print((char)b);
   }
     while(b!=-1);
       finp.close();
       }
   }
 }
```



In the section, we will learn how to write data to a file. As we have discussed, the FileOutputStream class is used to write data to a file.

Let us consider an example that writes the data to a file converting into the bytes.This program first check the existence of the specified file. If the file exist, the data is written to the file through the object

of the FileOutputStream class.

```
import java.io.*;
public class WriteFile{
   public static void main(String[] args) throws IOException{
    File f=new File("textfile1.txt");
    FileOutputStream fop=new FileOutputStream(f);
     if(f.exists()){
     String str="This data is written through the program";
        fop.write(str.getBytes());
       fop.flush();
       fop.close();
       System.out.println("The data has been written");
       }
        else
         System.out.println("This file is not exist");
   }
}
```

```
C:\WINDOWS\system32\cmd.exe

C:\html>javac WriteFile.java

C:\html>java WriteFile
The data has been written

C:\html>
```

CHECK YOUR PROGRESS - 2

1. Java supports ..... Standard Streams
2. In Java all files are ……..……..... oriented.
3. Java supports the ..... and .....I/O file streams
4. State whether true or false:
   a. You can write one complete output line to the console by calling the System.out.println() method.
   b. Java provides methods to read and write bytes from and to a file.

7.9 LET US SUM UP

- Java views each file as a sequential stream of bytes.
- Java uses streams to handle I/O operations through which the data is flowed from one location to another
- There are two types of streams Byte and Character
- Programs use byte streams to perform input and output of 8-bit bytes
- The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize.

- Java supports three Standard Streams: Standard Input,Standard Output,Standard Error
- Java provides a number of classes and methods that allow you to read and write files

7.10 ANSWER TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS -1

1. stream 2. java.io 3. Streams 4. Unicode 5. System
6. a) True b) true c) false

CHECK YOUR PROGRESS -2

1. three 2. Byte 3. FileInputstream, FileOutputStream
4. a) true b) true.

7.11 FURTHER READINGS

1. "*The Complete Reference, Java 2*", Tata McGraw-Hill Edition
2. "*JAVA How to Program*", Deitel & Deital, PHI Publication
3. You can alo visit the site www.java.sun.com

7.12 POSSIBLE QUESTIONS

1. What is stream in java? How Java represents a file?
2. How file and stream work in Java?
3. What are the two different types of streams? What are the advantages of character streams?
4. What is predefined stream?

UNIT – 8 INTRODUCTION TO APPLETS

UNIT STRUCTURE

8.1 LEARNING OBJECTIVES

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

- learn how to write applets in Java and their advantages and disadvantages in programming
- describe the life cycle of applet
- learn about event handling
- add Windows components to applets
- add audio file as well as animation to applets

8.2 INTRODUCTION

In the previous unit, we discussed the **io** package as well as the different **stream** classes in Java. Besides these we also presented how to read and write console input/output and handle files in Java.

Today internet has become an inevitable part of life. It is used for accessing libraries, getting information on latest happening, to transfer information, send email and communicate with others. Java has become a prime language today in making web pages interactive and user friendly. Java programs written to run on ***World Wide Web*** (WWW) are known as ***applets***. In this unit we will learn how to write applets and how to include them in web pages. The life cycle of applet will also be discussed in this unit. It is possible to change the colors, font of the text, set different background colors for different applets and make them look more attractive.

8.3 APPLETS AND THE WORLD WIDE WEB

An applet is a Java program that we can embed in a web page. Java applications are run by using a Java interpreter. Applets can run on any browser that supports Java. Applet can also be tested using the applet viewer tool included in the Java Development Kit. In order to run an applet it must be included in a web page, using HTML tags. Since Java's bytecode is platform independent, Java applets can be executed by browsers for many platforms, including Windows, Unix, Mac OS and Linux. When a Java technology-enabled web browser views a page that contains an applet, the applet's code is transferred to the clients system and executed by the browser's Java Virtual Machine (JVM).

Advantages of Applet :

- Applets are cross platform and can run on Windows, Mac OS and Linux platform
- Applets can work all the version of Java Plug-in
- Applets are supported by most web browsers
- Applets are cached in most web browsers, so will be quick to load when returning to a web page.
- User can also have full access to the machine if user allows.

Disadvantages of Applet :

- Java plug-in is required to run applet.
- Java applet requires JVM so first time it takes significant startup time.

- If applet is not already cached in the machine, it will be downloaded from internet and will take time.
- It is difficult to design and build good user interface in applets compared to HTML technology.

8.4 THE APPLET CLASS

The java.applet package is the smallest package in the Java API. The Applet class is contained in the java.applet package. Applet contains several methods that gives us detailed control over the execution of the applet. In addition, java.applet also define three interfaces: AppletContext, AppletStub, and AudioClip.

All applets are subclasses of Applet. Thus, all applets must import java.applet. applets must also import java.awt. all applets run in a window, it is necessary to include support for that window. Applets are not executed by the console-based Java run-time interpreter, they are executed by either a Web Browser or an applet viewer.

The Applet class contains a single default parameterless constructor, which is generally not used. Applets are constructed by the runtime environment when they are loaded and do not have to be explicitly constructed.

Applet provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods and methods that load and play audio clips.

8.5 APPLETS AND HTML

In order to run a Java applet, it is first necessary to have a web page that references the applet. The <Applet> tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. Applet tag takes zero or more parameters.

Syntax

<APPLET

CODE=”name of the class file that extends java.applet.Applet”

CODEBASE=”path of the class file” HEIGHT=”maximum height of the applet, in pixels”

WIDTH=”maximum width of the applet, in pixels”

VSPACE=”vertical space between the applet and the rest of the HTML”

HSPACE=”horizontal space between the applet and the rest of the HTML”

ALIGN=”alignment of the applet with respect to the rest of the web page”

ALT=”alternate text to be displayed if the browser does not support applets”>

<PARAM NAME=”parameter_name” value=”value_of_parameter”>

</APPLET>

The most commonly used attributes of the Applet tag are CODE, HEIGHT, WIDTH, CODEBASE and ALT. Also you can send parameter to the applet using the PARAM tag. The PARAM tag must be written between <APPLET> and </APPLET> .

CODE: this is required attribute that gives the name of the file containing your applet’s compiled .class file.

CODEBASE: this is an optional attribute, that specifies the base URL of the applet code.

HEIGHT AND WIDTH : these two are required attributes that gives the size in pixels of the applet display area.

VSPACE AND HSPACE: These attributes are optional. VSPACE specifies the space, in pixels, above and bellow the applet. HSPACE specifies the space, in pixels, on each side of the applet.

ALIGN: this is an optional attribute that specifies the alignment of the applet. The possible values are LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE and ABSBOTTOM.

ALT: this is also an optional attribute used to specify a short text message that should be displayed if the browser can not run the Java Applet.

PARAM NAME AND VALUE: the PARAM tag allows to specify applet specific arguments in an HTML page.

Example 1 : Simple Applet displaying HELLO WORLD

SimpleJavaApplet.java

```
import java.awt.Graphics; //This imports our Graphics class , which is used for drawing lines , squares, circles, text.

public class SimpleJavaApplet extends java.applet.Applet { //This means that our custom HelloWorld class extends the Applet class. That is , it is a subclass of Applet. Therefore we have access to all the methods of Applet, and we can extend our custom class to do further things.

public void paint(Graphics g) { //This method is our drawing method. This method draws anything that is in our applet to the screen. Note that we have passed an instance(g) of the Class Graphics to it.

g.drawString("HELLO WORLD", 5, 25); //drawString is a method of the Graphics class. We have an instance of the Graphics class - g. drawString takes a String and an x and a y coordinate.
        }
}
```

simpleapplet.html

```
<html>

<head>

<body>

 <APPLET CODE="SimpleJavaApplet.class" HEIGHT=250 WIDTH=250 >

 </APPLET>

 </body>

</html>
```

Both this file needs to stay in the same directory.

The HTML (*Hyper Text Marked up Language*) code tells the browser to load the compiled java applet SimpleJavaApplet.class that is in the same directory as this HTML file. It specifies the display area for the applet output as 250 pixels width and 250 pixels height.

To run an applet, we require one of the following tools:

a) Java Applet Viewer

b) Java enabled Web Brower (such as IE or Firefox)

Before running the program we need to compile the program. The next section demonstrate how to compile and run an applet in a graphical way. The example shows here are keep in a directory under c:\html.

a) Using Java Applet Viewer:

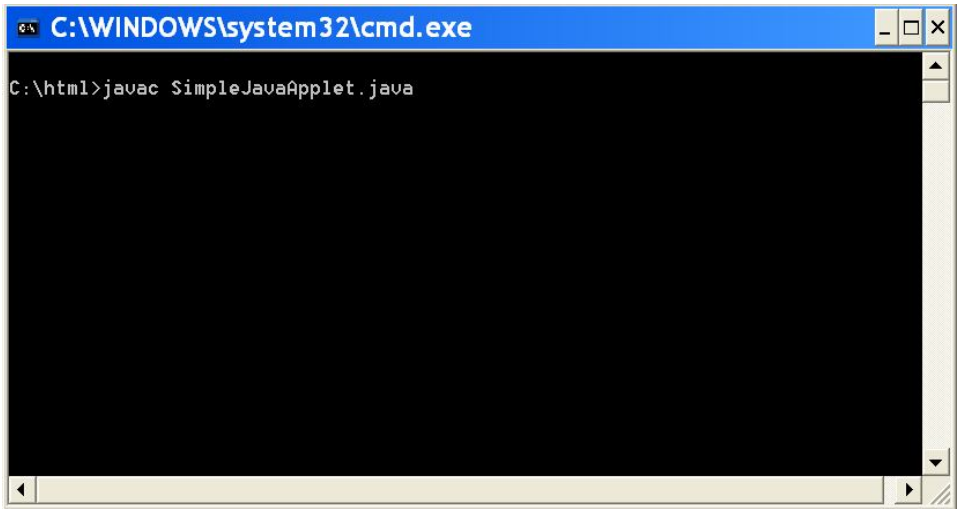


Fig: Compiling the Java program SimpleJavaApplet.java

The applet viewer is available as a part of the JDK. We can now run our applet as follows:

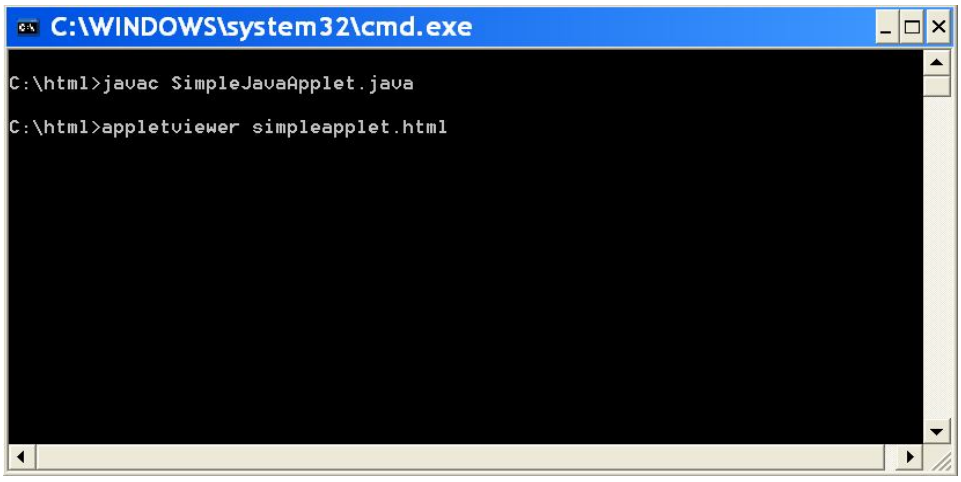


Fig: running applet using Applet Viewer.

OUTPUT:

When the browser encounters an <Applet> tag, it reserves a display area of the specified width and height for the class in the browser window, loads the bytecodes for the specified Applet subclass, creates an instance of the subclass and then calls for the instances init and start method.

b) Using Java enabled browser (Firefox)

Run the browser and then go to File->Open File , it will open a screen looks bellow. Select the file simpleapplet.htm and click on open.

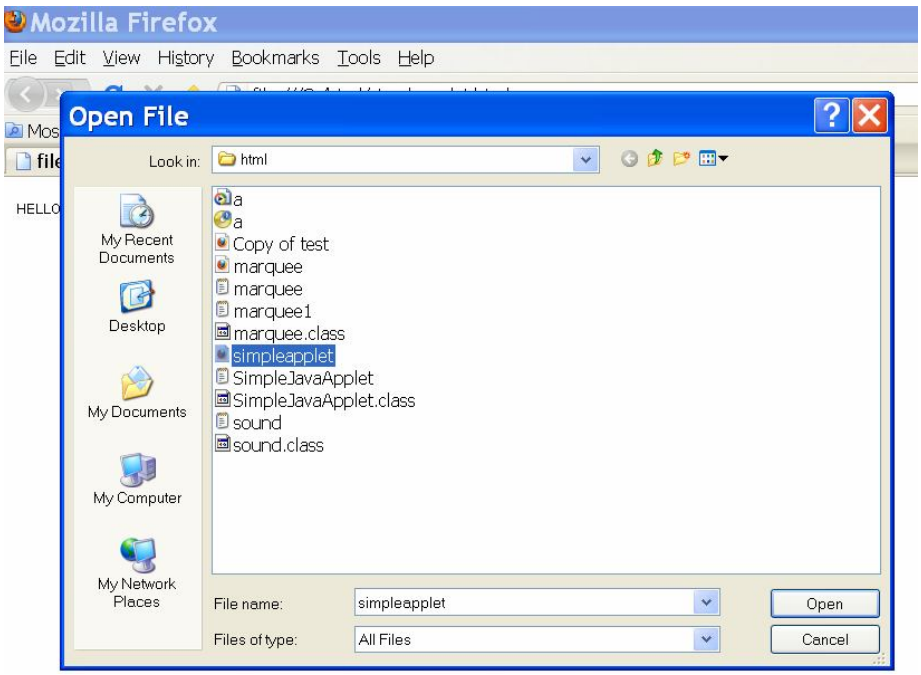
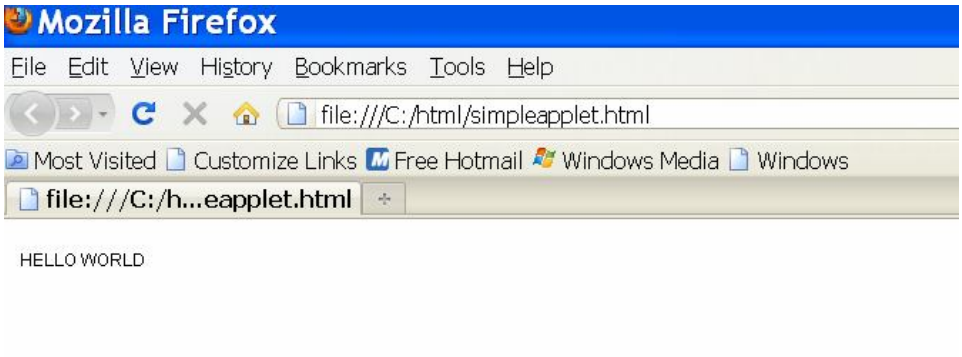


Fig: Select the html file that included the applet class

The applet is run on the browser and displays the word “HELLO WORLD” as bellow.



We can embed applets into Web pages in two ways.

- One way is that we can write our own applets and embed them into web pages. An applet developed locally and stored in a local system is known as a local applet.
- Secondly, we can download an applet from a remote computer system and then embed it into a web page. A remote applet is that which is developed by someone else and stored on a remote computer (web server) connected to the Internet. In order to locate and load a remote applet, we must know the applets address on the web. This address is known as the **URL** and must be specified in the applet’s HTML document as the value of the CODEBASE attribute.

Drawing Shapes Example:

In this program we will see how to draw the different types of shapes like line, circle and rectangle. There are different types of methods for the **Graphics** class of the *java.awt.*;* package have been used to draw the appropriate shape.

Here is the java code of the program:

CircleLine.java:

```
import java.applet.*;
import java.awt.*;
public class  CircleLine extends Applet{
```

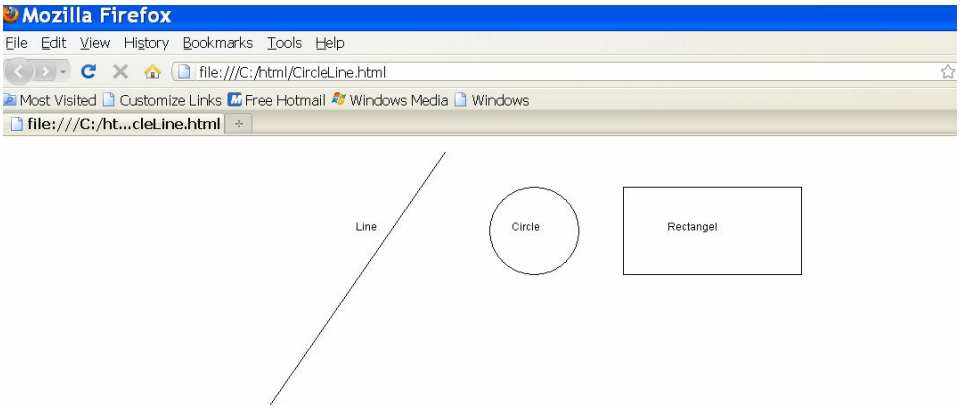
```
int x=300,y=100,r=50;
public void paint(Graphics g){
g.drawLine(3,300,200,10);      //  used to draw the line in the applet.
g.drawString("Line",100,100); // draws the given string as the parameter
g.drawOval(x-r,y-r,100,100); //  draws the circle
g.drawString("Circle",275,100); // draws the given string as parameter
g.drawRect(400,50,200,100); // draws the rectangle
g.drawString("Rectangel",450,100); // draws the given string as parameter
  }
}
```

Here is the HTML code of the program:

CircleLine.html:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<div align="center">
<APPLET CODE="CircleLine.class" WIDTH="800" HEIGHT="500"></
APPLET>
</BODY>
</HTML>
```

Output :



CHECK YOUR PROGRESS -1

1. Fill in the blanks:
   i. The ..... packages is the smallest package in the Java API.
   ii. ..... is used to execute an Applet.
   iii. The Applet tag is written the .....of an HTML document.
2. State whether the following statements are true or false:
   a. Applet can be run using Applet Viewer.
   b. Applet are not supported by web browser.
   c. Applet can run on Linux platform.

8.6 THE LIFE CYCLE OF AN APPLET

In this section we will learn about the lifecycle of an applet. Applet runs in the browser and its lifecycle method are called by JVM when it is loaded and destroyed. The lifecycle of an Applet are:

- init(): This method is called to initialized an applet
- start(): This method is called after the initialization of the applet.
- stop(): This method can be called multiple times in the life cycle of an Applet.
- destroy(): This method is called only once in the life cycle of the applet when applet is destroyed.

init () method:

The life cycle of an applet is begin on that time when the applet is first loaded into the browser and called the init() method. The init() method is called only one time in the life cycle on an applet. The init() method is basically called to read the PARAM tag in the html file. The init () method retrieve the passed parameter through the PARAM tag of html file using get Parameter() method All the initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method .After the initialization of the init() method user can interact with the Applet and mostly applet contains the init() method.

Start () method:

The start method of an applet is called after the initialization method init(). This method may be called multiples time when the Applet needs to be started or restarted. For Example if the user wants to return to the Applet, in this situation the start() Method of an Applet will be called by the web browser and the user will be back on the applet. In the start method user can interact within the applet.

Stop () method:

The stop() method can be called multiple times in the life cycle of applet like the start () method. Or should be called at least one time. There is only miner difference between the start() method and stop () method. For example the stop() method is called by the web browser on that time When the user leaves one applet to go another applet and the start() method is called on that time when the user wants to go back into the first program or Applet.

destroy() method:

The destroy() method is called only one time in the life cycle of Applet like init() method. This method is called only on that time when the browser needs to Shut down.

The following diagram depicts the life cycle of an applet

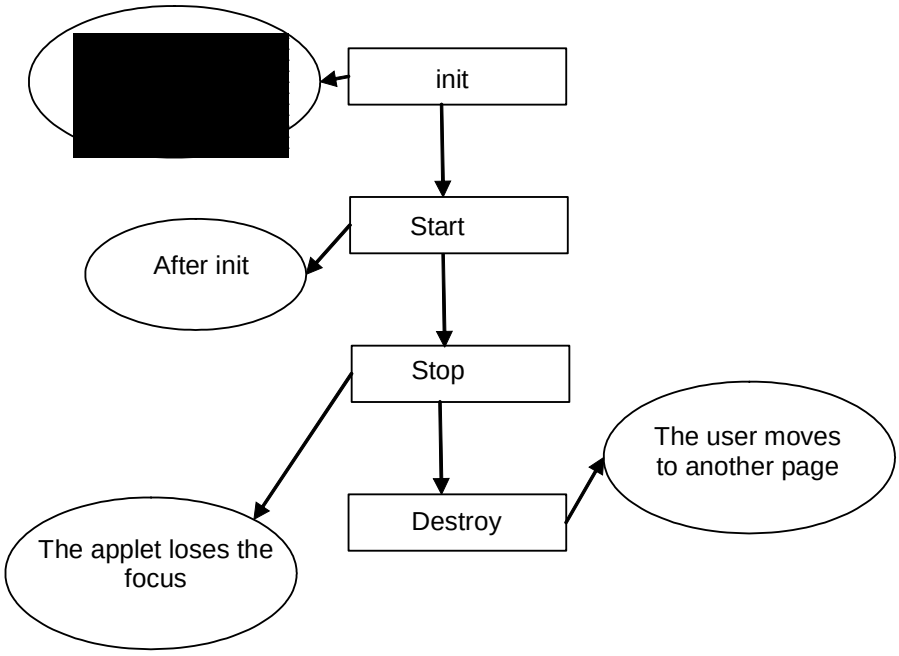


Fig: Life Cycle of An APPLET

Loading the Applet :

As a result of the applet being loaded, you should see the text "initializing... starting...". When an applet is loaded, here's what happens:

* An instance of the applet's controlling class (an Applet subclass) is created.
* The applet initializes itself.
* The applet starts running.

Leaving and Returning to the Applet's Page :
When the user leaves the page, for example, to go to another page, the browser stops and destroys the applet. The state of the applet is not preserved. When the user returns to the page, the browser intializes and starts a new instance of the applet.

Reloading the Applet :
When you refresh or reload a browser page, the current instance of the applet is stopped and destroyed and a new instance is created.

Quitting the Browser:
When the user quits the browser, the applet has the opportunity to stop itself and perform a final cleanup before the browser exits.

CHECK YOUR PROGRESS - 2

1. The ..... method is called the first time an applet is loaded into the memory of a computer.
2. The ..... method is called by the browser when the user moves to another page.

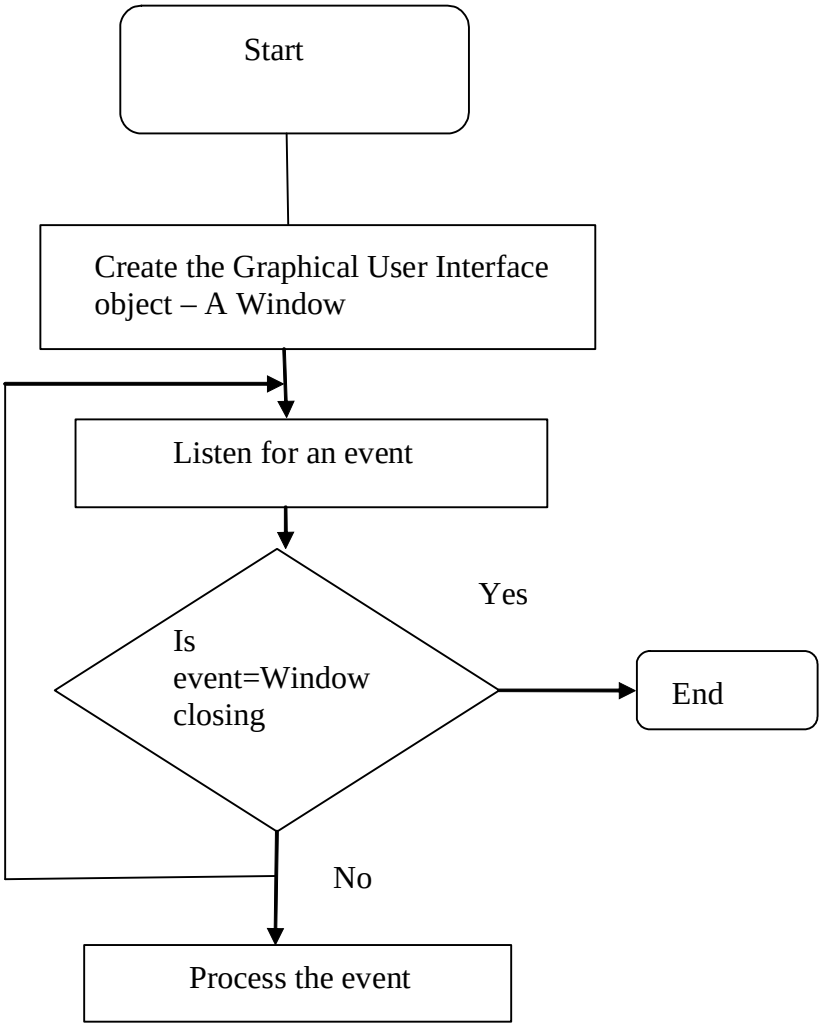
8.7 EVENT HANDLING

Event handling is essential to GUI programming. The program waits for a user to perform some action. The user controls the sequence of operations that the application executes through a GUI. This approach is called event driven programming.

- **Components of an Event**

An event comprises of three components.

- **Event Object-** When a user interacts with a application by clicking a mouse or pressing a key from the keyboard, an event is generated. The operating system keep track of this event and the data associated with it. For example the time at which the event occurred, the event type (like a mouse click).
- In Java events are represented by objects that describes the events themselves. Java has a number of classes that describes and handle different categories of event.
- **Event Source-** An event source ia an object that generates an event. For example if you click on a button, an ActionEvent object is generated. The object of the ActionEvent class contains information about the event.
- **Event-handler-** An event handler is a method that understands the event and processes it. The event-handler method takes an Event object as a parameter



- **Event Classes**

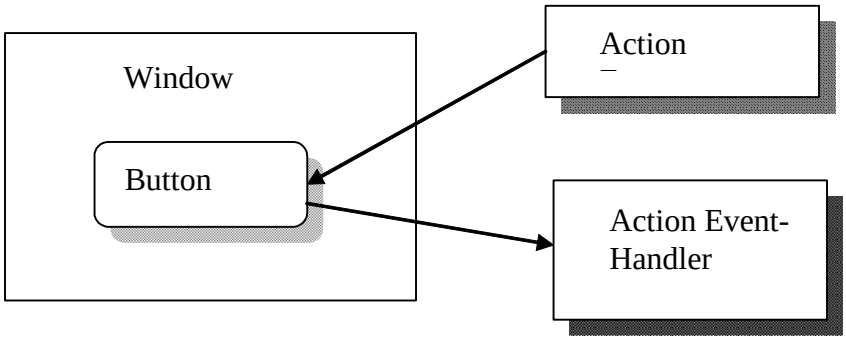
The EventObject class is at the top of the event class hierarchy. It belongs to the java.util package. Most other event classes are present in the java.util.event package.

- **Event Listener**

An object delegates the task of handling an event to an event listner. When an event occurs, an event object of the appropriate type is created. This object is passed to the listener. A listener must implement the interface that has the method for event-handling. A component can have multiple listeners.

- **Event-Handling**

When an event occurs, it is sent to the component from where the event originated. The component registers a listener, which contains event-handler. Event-handler receives and process events.



Every event has a corresponding listener interface that specifies the methods that are required to handle the event. Event object are reported to registered listener. To enable a component to handle events, you must register an appropriate listener for the components.

8.8 USING WINDOW COMPONENT

Window components can be added to the applet to create a friendly user interface. Java provides a number of window components, which can be found in the java.awt library.

All of the window components added to an applet are confined to the applet window area in the current browser document. The applet window contains the entire paintable region; anything that does not fit inside that region is clipped.

Using the provided window components, applets have the capability to contain the same look and feel of other window application programs. Applets can use all GUI components that can be used by Panel objects. In addition, applets have access to many of the capabilities provided by the browser. In the next section this functionality is discussed in details.

8.9 ADDING AUDIO AND ANIMATION

Adding Audio

The Applet class provides the capability to play audio files. The play() method of the Applet class can be used to play an audio file that is identified by an URL. A more flexible approach is to load an object that implements the AudioClip interface and then invoke the object's play(), loop(), and stop() methods. The getAudioClip() method can be used to load an audio file by identifying its URL.

The example describe bellow plays the testsound once and continues in an endless loop playing the testingsound sound clip.

sound.java

.au = Sun audio format

```
import java.applet.AudioClip;  // This allows us to use the Audioclip and
play methods in the Applet class of the Java Abstract Windowing toolkit.

public class sound extends java.applet.Applet { // We want to generate an
applet .

public void init(){                    // initialization

play(getCodeBase(),"testsound.au");  // Play the testsound directly once.
The file is in the same directory as the applet.

AudioClip clip = getAudioClip (getCodeBase(), "testingsound.au"); //
Declare clip as an instance of the class AudioClip and loop to repeat the
call of the testingsound. This sound file is also in the same directory as
the applet itself.
clip.loop();
 }


}
```

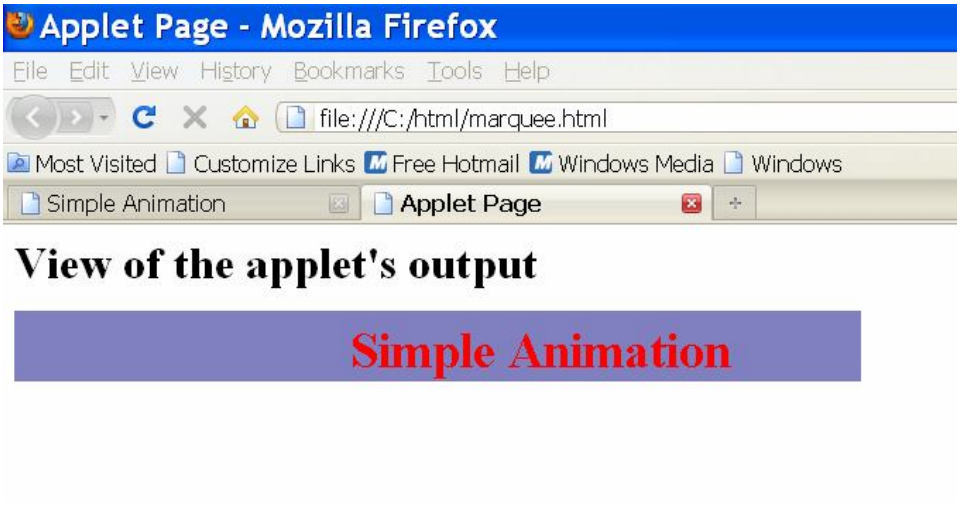
Sound.html

```
<html>
<head>
<body>
 <APPLET CODE=”sound.class” HEIGHT=150 WIDTH=150>
 You can not see this brilliant Java Applet.
 </APPLET>
 </body>
</html>
```

To run this code we need to type this code in a text editor and save the file as “sound.java”. After compiling this program we need to create a HTML file with a reference to the class file. Save the HTML file in the same directory as the class file. Also the 2 sound file needs to place on the same directory.

Adding animations

We can also create animations using Applet code. Let us take one simple example of scrolling marquee. The animation needs to do three things. Draw the string, wait for some length of time and then change the location where the string is to be drawn to the left one pixel. Repeating this process will appear to animate the string from right to left. The output of the program is shown bellow and the text will scroll from right to left.



The code of the above output is given bellow:

```
import java.awt.Graphics;                    //include the Graphics font
import java.awt.Color;                       and color capabilities
import java.awt.Font;
import java.awt.Image;

public class marquee extends                 //create an applet that is
java.applet.Applet implements                'runnable'
Runnable                                     allowing multithreading
{
String mesag ="Simple Animation";            //These Declarations set up
Font mfont = new                             the string that will scroll the
Font("TimesRoman",Font.BOLD, 36);            font and size The x
int Xposition = 600;                         coordinate
Image scrnBuf;
Graphics scrnG;

Thread runner;                               //a thread to control the flow
                                             of the program.
public void init()                           The thread will not start until
{                                            the parent thread calls this
scrnBuf = createImage(600,50);               Thread. The new thread
scrnG = scrnBuf.getGraphics();               begins executing the run
}                                            method of our runnable
                                             class.
public void start()
 {
  if (runner == null);
  {
    runner = new Thread(this);
    runner.start();
  }
 }
```

```
public void stop()                    //This stops the thread from
 {                                    executing. Returning from
  if (runner != null);                the run method causes an
  {                                   automatic call to stop
    runner.stop();
    runner = null;

  }
 }
public void run()                     //Control passes here as
{                                     soon as the thread starts
  while(true)                         When this method returns
  {                                   the thread stops
  repaint();                          We simply set the
  try {Thread.sleep(30);}             background color Paint the
  catch(InterruptedException e) { }   screen with the paint method
  }                                   Put the thread to sleep and
}                                     check for any errors


public void update(Graphics g)        //The paint method actually
{                                     places the text on the screen
  paint(g);                           First set the font Draw the
}                                     string at the current x
public void paint(Graphics g)         position and decrement the
{                                     x position If the text has
  Color c = new Color(128,128,192);   scrolled off the screen reset
  scrnG.setColor(c);                  the x position to start the
  scrnG.fillRect(0,0,600,50);         process over on the right
  scrnG.setColor(Color.red);          side
  scrnG.setFont(mfont);
```

```
scrnG.drawString(mesag,Xposition,40);
  Xposition--;
  if (Xposition < -290)
    {
    Xposition = 600;
    }
  g.drawImage(scrnBuf, 0 , 0 , this);
}

}
```

8.10 LET US SUM UP

- An Applet runs in a webpage.
- The Applet class is the only class of the java.applet package.
- An Applet can be executed using the appletviewer or a Java-enabled browser.
- The Applet tag is used to embed an applet in a web page.
- The init() method is called the first time the applet is loaded into the memory of a computer.
- The start() method is called immediately after the init() method and every time the applet receives focus .
- The stop() method is called every time the user moves on to another web page.
- The destroy() method is called just before the browser is shut down.
- The components of an event are: Event Object, Event Source, Event Handler.

8.10 ANSWER TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS -1

1. (i)Java.applet (ii) Appletviewer (iii) Body tag
2. a) True b) False c) True

CHECK YOUR PROGRESS -2

1. init 2. destroy

8.4 FURTHER READINGS

1. "*The Complete Reference, Java 2*", Tata McGraw-Hill Edition
2. www.java.sun.com

8.5 POSSIBLE QUESTIONS

1. Explain the life cycle of an appet?
2. What are the components of Event Handler?
3. What are the advantages and disadvantages of APPLET?

UNIT - 9 AWT AND SWINGS

UNIT STRUCTURE

9.1 LEARNING OBJECTIVES

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

- learn about the concept of AWT and its components
- describe event handling
- learn about Swing and its components
- display test and images in a window
- learn about layout manager

9.2 INTRODUCTION

In this unit we will learn about the **AWT** package of the Java and a brief description of **Swing**. The AWT is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is now part of the J**ava Foundation Classes** (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program. Here we will provide a

brief description about different AWT components, event handling in AWT and Swing. At the end we will learn about Layout Manager.

9.3 AWT BASICS

AWT stands for ***Abstract Windowing Toolkit***. It contains all classes to write the program that interface between the user and different windowing toolkits. We can use the AWT package to develop user interface objects like buttons, checkboxes, radio buttons and menus etc.

Now a days developer are using Swing components instead of AWT to develop good GUI for windows applications.

9.4 AWT COMPONENTS

In this section we will learn about the different components available in the Java AWT package for developing user interface for our program. Following are some of the components of Java AWT:

Labels: This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in our application and label never perform any type of action. Syntax for defining the label is:

Label label_name = new Label (“This is the label text”);

Above code simply represents the text for the label.

Label label_name = new Label (“This is the label text.”, Label.CENTER);

label can be left, right or centered. Above declaration used the center justification of the label using the *Label.CENTER*

Buttons: This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for our application. The syntax of defining the button is as follows:

Button button_name = new Button (“This is the label of the button.”);
We can change the Button’s label or get the label’s text by using the Button.setLabel(String) and Button.getLabel() method. Buttons are added to its container using the add (button_name) method.

Check Boxes: This component of Java AWT allows us to create check boxes in our applications. The syntax of the definition of Checkbox is as follows:

CheckBox checkbox_name = new Checkbox (“Optional check box 1”, false);

Above code constructs the unchecked Checkbox by passing the Boolean valued argument false with the Checkbox label through the Checkbox() constructor. Defined Checkbox is added to its container using add (checkbox_name) method. We can change and get the checkbox’s label using the setLabel (String) and getLabel() method. We can also set and get the state of the checkbox using the setState(boolean) and getState() method provided by the Checkbox class.

Radio Button: This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes which group name is same. Only one Checkbox from a Checkbox Group can be selected at a time. Syntax for creating radio buttons is as follows :

CheckboxGroup chkgp = new CheckboxGroup();

add (new Checkbox (“One”, chkgp, false);

add (new Checkbox (“Two”, chkgp, false);

add (new Checkbox (“Three”,chkgp, false);

In the above code we are making three check boxes with the label “One”, “Two” and “Three”. If we mention more than one true valued for checkboxes then our program takes the last true and show the last check box as checked.

Text Area: This is the text container component of Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

TextArea txtArea_name = new TextArea();
We can make the Text Area editable or not using the setEditable (Boolean) method. If we pass the Boolean valued argument false then the text area

will be non-editable otherwise it will be editable. The text area is by default in editable mode. Text are set in the text area using the setText(string) method of the TextArea class.

Text Field: This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows:

TextField txtfield = new TextField(20);

We can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

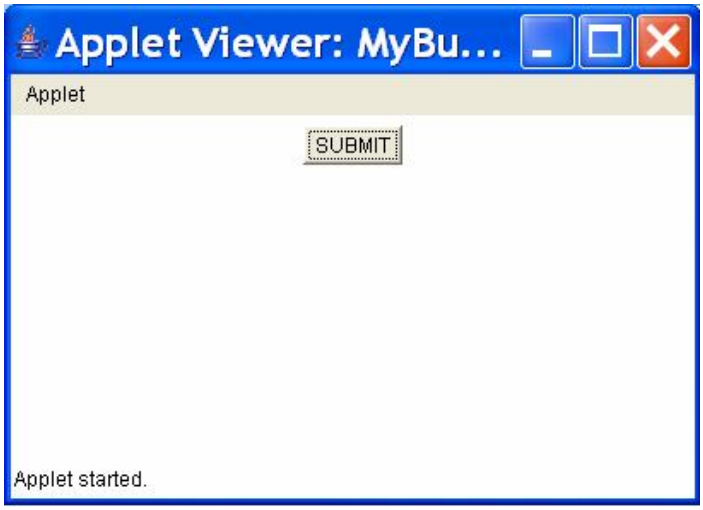
As shown in the example below, a button is represented by a single label. That is the label shown in the example can be pushed with a click of a mouse.

MyButton.java

```
import java.awt.*;
import java.applet.Applet;
public class MyButton extends Applet
 {
    public void init()
    {
          Button button = new Button(“SUBMIT”);
           add(button);
     }
}
```

Here is the HTML code:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<APPLET ALIGN=”CENTER” CODE=”MyButton” WIDTH=”400"
HEIGHT=”200"></APPLET>
</BODY>
</HTML>
```



CHECK YOUR PROGRESS -1

1.Fill in the blanks :

i. ……..……is generally used to show the text or string in your application.

ii. …….…… never perform any type of action

iii. ……..…….. component of Java AWT allows you to create check boxes in your applications

iv. ……..…….. is the special case of the Checkbox component of Java AWT package.

v. ……..…… and ……..…….. is the text container component.

9.5 EVENT HANDLING

There are many types of events that are generated by our AWT Application. These events are used to make the application more effective and efficient. Generally, there are twelve types of event are used in Java AWT. These are as follows:

1. **ActionEvent** : It indicates the component-defined events occurred i.e. the event generated by the component like Button, Checkboxes etc.

2. **AdjustmentEvent** : This is the AdjustmentEvent class extends from the AWTEvent class. When the Adjustable Value is changed then the event is generated.

3. **ComponentEvent** : This is the low-level event which indicates, if the object moved, changed and its states (visibility of the object). This class only performs the notification about the state of the object.

4. **ContainerEvent** : This is a low-level event which is generated when container's contents changes because of addition or removal of a components.

5. **FocusEvent** : This indicates about the focus where the focus has gained or lost by the object

6. **InputEvent** : This event class handles all the component-level input events. This class acts as a root class for all component-level input events.

7. **ItemEvent** : The ItemEvent class handles all the indication about the selection of the object i.e., whether selected or not.

8. **KeyEvent** : It handles all the indication related to the key operation in the application if we press any key for any purposes of the object then the generated event gives the information about the pressed key. These types of events check whether the pressed key left key or right key, 'A' or 'a' etc.

9. **MouseEvent** : It handle all events generated during the mouse operation for the object. That contains the information whether mouse

is clicked or not if clicked then checks the pressed key is left or right.

10. **PaintEvent** : The PaintEvent class only ensures that the paint() or update() are serialized along with the other events delivered from the event queue.

11. **TextEvent** : TextEvent is generated when the text of the object is changed.

12. **WindowEvent** : If the window or the frame of our application is changed (Opened, closed, activated, deactivated or any other events are generated), WindowEvent is generated.

9.6 INTRODUCTION TO SWING

The Java ***Swing*** provides the multiple platform independent APIs interfaces for interacting between the users and GUIs components. Java provides an interactive feature for design the GUIs toolkit or components like: *labels, buttons, text boxes, checkboxes, combo boxes, panels* and *sliders* etc. All AWT flexible components can be handled by the Java Swing. The Java Swing supports the plugging between the look and feel features. The look and feel that means the dramatically changing in the component like JFrame, JWindow, JDialog etc. for viewing it into the several types of window.

9.7 SWING COMPONENTS

There are many components which are used for the building of GUI in Swing. The Swing Toolkit consists of many components for the building of GUI. These components are also helpful in providing interactivity to Java applications. Following are the some of the components which are included in Swing toolkit:

1. list controls
2. buttons
3. labels
4. tree controls
5. table controls

All AWT flexible components can be handled by the Java Swing. Swing toolkit contains far more components than the simple component toolkit. In the next section we are going to show some examples.

Text Field: The following example shows how to create a text field. The swing text field is encapsulated by the JTextComponent class which extends JComponent. One of its subclass JTextField allows to edit one line of text box.

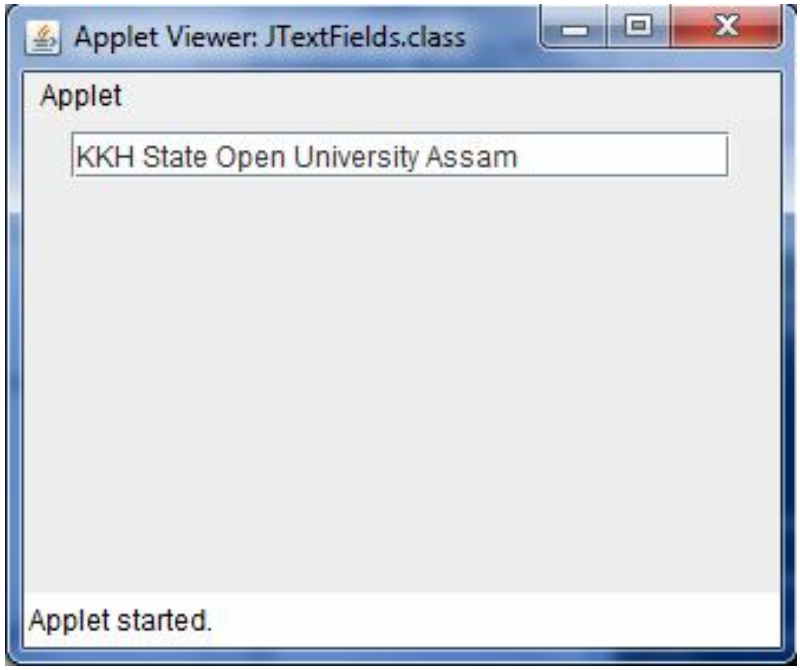
JTextFields.java:

```
import java.awt.*;
import javax.swing.*;

public class JTextFields extends JApplet
{
        JTextField jtf;
        public void init()
        {
         //Get content pane
         Container contentPane=getContentPane();
         contentPane.setLayout(new FlowLayout());

         //Add Text field
         jtf= new JTextField(25);
         //add Textbox to the content pane
         contentPane.add(jtf);
        }
}
```

JTextField.html

```
 <html>

 <applet code="JTextFields.class" height=200 width=320>

 </applet>
</html>
```

Output:



Buttons:

Swing buttons are subclasses of the AbstructButton class, which extends JComponent. The JButton class provides the functionality of a push button. The following example shows a push button.

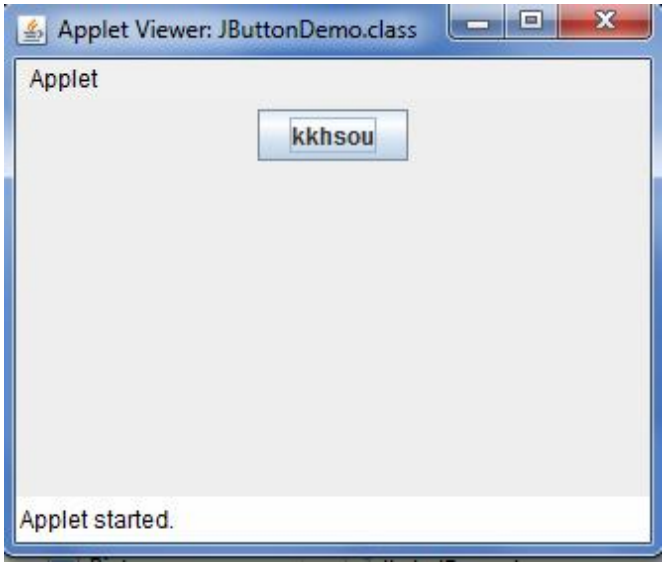
JButtonDemo.java:

```
import java.awt.*;
import javax.swing.*;

public class JButtonDemo extends JApplet{

        public void init(){
                //Get content pane
                Container contentPane=getContentPane();
                contentPane.setLayout(new FlowLayout());
                //Add button to the content pane
                JButton jb=new JButton("kkhsou");
                contentPane.add(jb);
        }
}
```

JButton.html:

```
<html>
        <applet code="JButtonDemo.class" height=200 width=320>
        </applet>
</html>
```

Output:



Check Boxes: The JCheckBox class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. The following example shows how to create an applet that displays three check boxes.

JCheckBoxDemo.java

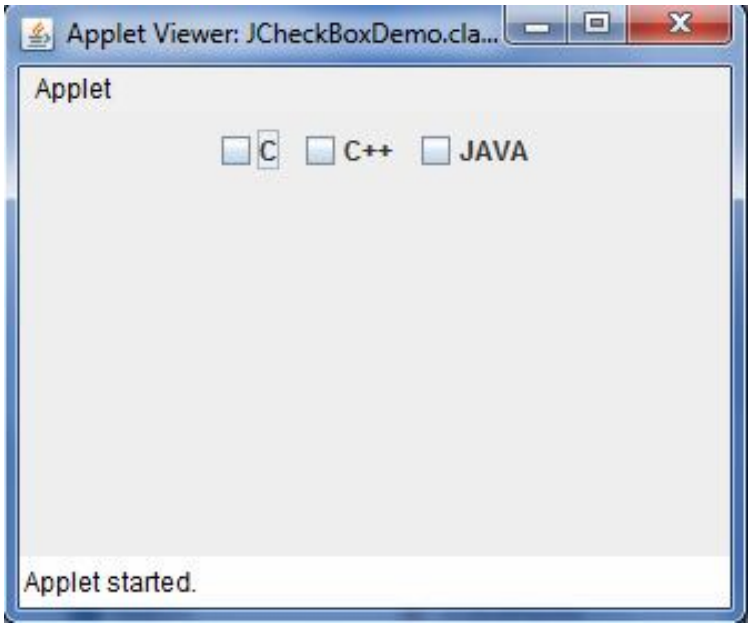
```
import java.awt.*;
import javax.swing.*;

public class JCheckBoxDemo extends JApplet
{
        public void init()
        {
```

```
                    //Get content pane
                    Container contentPane=getContentPane();
                    contentPane.setLayout(new FlowLayout());
                    //Add checkbox to the content pane
                    JCheckBox cb=new JCheckBox("C");
                    contentPane.add(cb);
                    cb=new JCheckBox("C++");
                    contentPane.add(cb);
                    cb=new JCheckBox("JAVA");
                    contentPane.add(cb);
                }
            }
```

JCheckBox.html:

```
<html>
        <applet code="JCheckBoxDemo.class" height=200 width=320>
        </applet>
 </html>
```

Output:



9.9 EVENT HANDLING

Events are an important part in any GUI program. All GUI applications are event-driven. An application reacts to different event types which are generated during its life. Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g., internet connection, window manager, timer. In the event model, there are three participants:

- event source
- event object
- event listener

The ***Event source*** is the object whose state changes. It generates Events. The ***Event object*** (Event) encapsulates the state changes in the event source. The ***Event listener*** is the object that wants to be notified. Event source object delegates the task of handling an event to the event listener. Event handling in Java Swing toolkit is very powerful and flexible. Java uses

Event Delegation Model. We can specify the objects that are to be notified when a specific event occurs.

Event object: When something happens in the application, an event object is created. For example, when we click on the button or select an item from list. There are several types of events. An ActionEvent, TextEvent, FocusEvent, ComponentEvent etc. Each of them is created under specific conditions. Event object has information about an event, that has happened.

9.9 DISPLAY TEXT AND IMAGE

The following example shows how to create and display a label consisting of both text and image. The applet started by getting ist content pane. Then an ImageIcon object created for the file kkhsou_logo.jpg. On the JLabel constructor first argument is text, seconf argument is ImageIcon object and the third argument is alignment. The align argument is LEFT, RIGHT, CENTER, LEADING or TRAILING. Finally the label is added to the content pane.

JLabelDemo.java

```
import java.awt.*;
import javax.swing.*;
public class JLabelDemo extends JApplet{
        public void init(){
         //Get content pane
         Container contentPane=getContentPane();
         //create icon
         ImageIcon ii=new ImageIcon("kkhsou_logo.jpg");

         //create label
JLabel jl=new JLabel("KKH Open University", ii, JLabel.CENTER);
         //add label to the content pane
         contentPane.add(jl);
        }
}
```

JLabelDemo.html

```
<html>
<applet code="JLabelDemo.class" height=200 width=320>
</applet>
</html>
```

Output

9.10 LAYOUT MANAGER

To create layouts, we use layout managers. Layout managers are one of the most difficult parts of modern GUI programming. We can use no layout manager, if we want. There might be situations, where we might not need a layout manager. But to create truly portable, complex applications, we need layout managers. Without layout manager, we position components using absolute values.

There are some of the common tasks associated to use layout managers:

- Setting Layout Manager
- Adding Components to a Container
- Providing Size and Alignment Hints
- Putting Space Between Components
- Setting the Container's Orientation
- Tips on Choosing a Layout Manager
- Third-Party Layout Managers

In Java a layout manager class implements the LayoutManager interface. It is used to determine the position and size of the components within a container. Components can provide size and alignment hints, still the container's layout manager has the final authority on the size and position of the components within the container.

FlowLayout manager : This is the simplest layout manager in the Java Swing toolkit. It is mainly used in combination with other layout managers. When calculating its children size, a flow layout lets each component assume its natural (preferred) size.

The manager puts components into a row. In the order, they were added. If they do not fit into one row, they go into the next one. The components can be added from the right to the left or vice versa. The manager allows aligning the components.

GridLayout: The GridLayout layout manager lays out components in a rectangular grid. The container is divided into equally sized rectangles. One component is placed in each rectangle.

BorderLayout: A BorderLayout manager is a very handy layout manager. It divides the space into five regions. *North, West, South, East* and *Centre*. Each region can have only one component. If we need to put more components into a region, we can simply put a panel there with a manager of our choice. The components in N, W, S, E regions get their preferred size. The component in the centre takes up the whole space left.

BoxLayout: BoxLayout is a powerful manager that can be used to create sophisticated layouts. This layout manager puts components into a row or into a column. It enables nesting, a powerful feature, which makes this manager very flexible. It means that we can put a box layout into another box layout.

CHECK YOUR PROGRESS -2

1.Fill in the blanks:

i. ....., the event generated by the component like Button, Checkboxes etc.

ii. ……..…… is the low-level event which indicates, if the object moved, changed and it's states

iii. ……..…….. handle all events generated during the mouse operation for the object.

iv. Events are an important part in any ..... program.

v. The ..... layout manager lays out components in a rectangular grid.

vi. ..... can be used to create sophisticated layouts.

vii. a layout manager class implements the .....interface.

9.11 LET US SUM UP

- AWT stands for Abstract Windowing Toolkit
- You can use the AWT package to develop user interface objects like buttons, checkboxes, radio buttons and menus etc.
- Some components of Java AWT are Labels, Buttons, Check Boxes, Radio Button, Text Area, Text Field.
- There are many types of events that are generated by your AWT Application. Generally, there are twelve types of event are used in Java AWT.
- The Java Swing provides the multiple platform independent APIs interfaces for interacting between the users and GUIs components.
- The Swing Toolkit consists of many components for the building of GUI.
- All AWT flexible components can be handled by the Java Swing.
- Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g. internet connection, window manager, timer.
- Event handling in Java Swing toolkit is very powerful and flexible. Java uses Event Delegation Model.
- Layout managers are one of the most difficult parts of modern GUI programming

9.12 ANSWER TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS – 1

1. i) Label ii) label iii)check boxes
   iv)radio button v)text area , text field

CHECK YOUR PROGRESS – 2

1. i)ActionEvent ii)Component Event
   iii)MouseEvent v)GUI
   v)GridLayout vi)BoxLayoutv
   vii)LayoutManager

9.13 FURTHER READINGS

1. "*The Complete Reference, Java 2*", Tata McGraw-Hill Edition
2. "*JAVA How to Program*", Deitel & Deital, PHI Publication
3. You can visit the site www.java.sun.com

9.14 POSSIBLE QUESTIONS

1. What is AWT? List 5 AWT components.
2. Briefly discuss about three Layout manager.
3. What are the two different types of event used in Java AWT?
4. Describe some components of Java AWT.
5. What do you mean by event handling? What are the types of eve6nts used in Java AWT?

UNIT - 10 INTRODUCTIONS TO JDBC

UNIT STRUCTURE

10.1 LEARNING OBJECTIVES

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

- learn about the database connectivity in Java,- API
- learn about different types of JDBC Drivers
- describe Client Server architecture
- define data source
- learn about resultset
- learn about prepared statement and callable statement

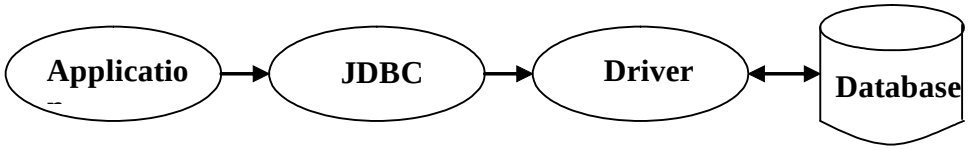
10.2 INTRODUCTION

In the earlier unit we became familiar with the AWT and its different components. We also learn how to handle an event, introduction of SWING and its components.

This unit describes the Java database connectivity. It also introduced the different types of JDBC drivers and a brief introduction of client server modal.

10.3 BASIC STEPS TO JDBC

Java provides a mechanism for handling database known as ***JDBC*** (***Java Database Connectivity***). Using JDBC it is possible to communicate with a wide variety of database management system using SQL. JDBC is a programming interface between Java programs and database management system.



Java application calls the JDBC library. JDBC loads a driver which talks to the database. We can change database engines without changing database code.

A program uses a Java class known as JDBC driver to connect to a database. A special JDBC driver, known as JDBC-ODBC bridge, makes it possible to a vast number of ODBC drivers usable from JDBC.

The main advantage of JDBC is it provides a standard interface to all database management system.

There are four basic steps required to work with JDBC. The four steps are as follows:

1. Load a JDBC driver for your DBMS. This involves a statement Class.forName(), specifying the driver class name.
2. Use the driver to open a connection to a particular database.

3. Issue SQL statements through the connection. Once the connection established, it can be used to create objects through which SQL commands can be made.
4. Process result sets required by the SQL operations.

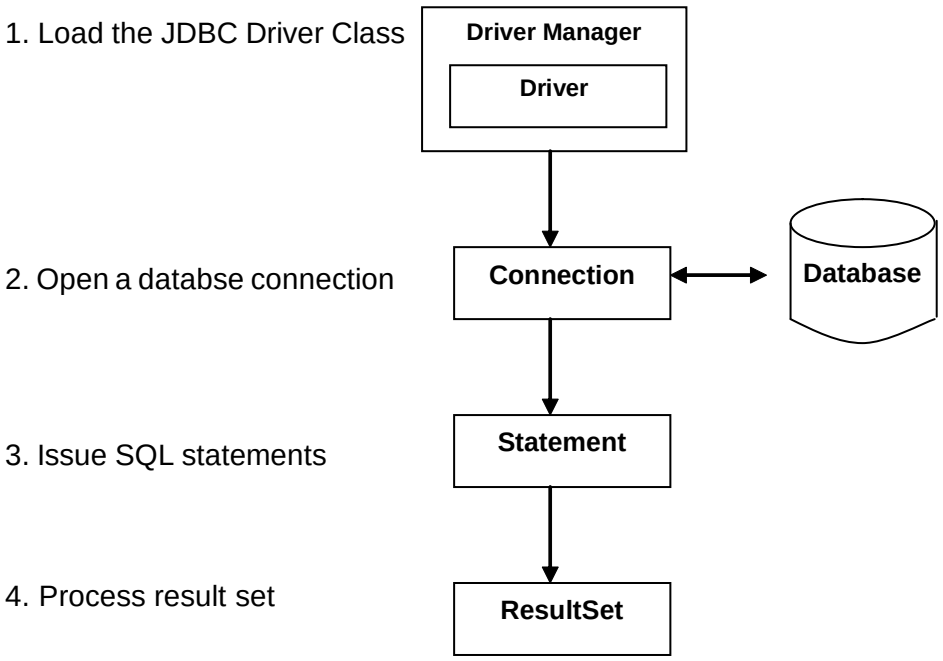


Fig.10.1 : Basic steps involved in JDBC operations

10.4 API

An ***application programming interface*** (***API***) is an interface implemented by a software program to enable interaction with other software, much in the same way that a user interface facilitates interaction between humans and computers.

The Java API is the set of classes included with the Java Development Environment. These classes are written using the Java language and run on the JVM. The Java API includes everything from collection classes to GUI classes.

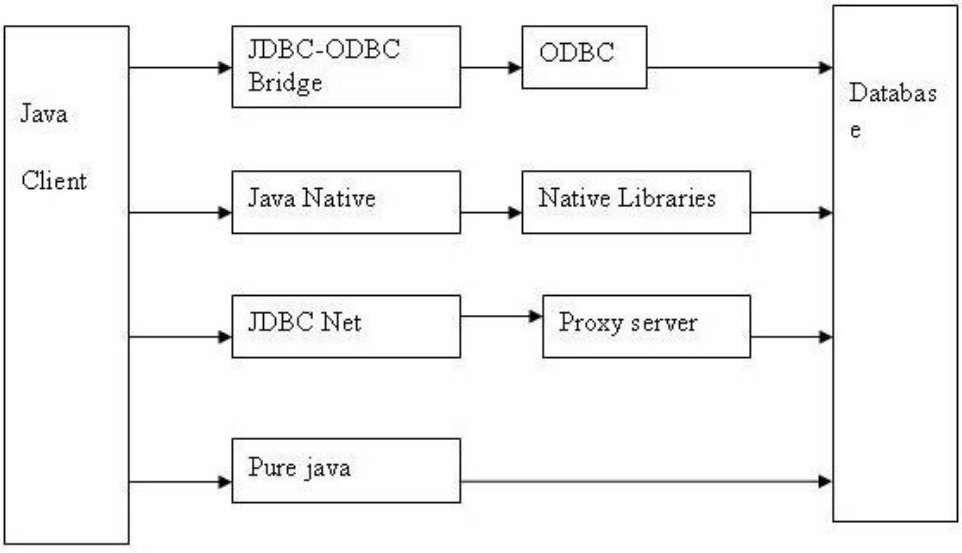
The JDBC API is a Java API for accessing virtually any kind of tabular data. The JDBC API consists of a set of classes and interfaces written in the Java programming language that provide a standard API for tool/database developers and makes it possible to write industrial-strength database applications entirely in the Java programming language.

The JDBC API makes it easy to send SQL statements to relational database systems and supports all dialects of SQL. But the JDBC API goes beyond SQL, also making it possible to interact with other kinds of data sources, such as files containing tabular data.

10.5 JDBC DRIVERS

A driver is Java class, usually supplied by the database vendor, which implements the java.sql.Driver interface. The primary function of the driver is to connect to a databse and return a connection object.

There are four types of driver classified according to their architecture.



Type 1 - JDBC-ODBC bridge: This type of driver connect to database through an intermediate ODBC driver. Several drawbacks are involved with this approach and so it is used where there is no other driver is available.

Advantage

1. The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

Type 2 – Native API: This type of driver use native method to call vendor specific API functions.

Advantage

The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type1 and also it uses Native api which is Database specific.

Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native api as it is specific to a database
4. Mostly obsolete now
5. Usually not thread safe.

Type 3 –Pure Java to database middleware: This driver communicates using a network protocol to a middleware server, which in turn, communicates to one or more database management systems.

Advantage

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.

3. There are many opportunities to optimize portability, performance, and scalability.
4. The protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.
7. They are the most efficient amongst all driver types.

Disadvantage

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.

Type 4 – Pure Java direct to database: This type of driver call directly.

Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

10.6 JDBC CONNECTION MANAGEMENT

A Connection is the session between your java program and database. whenever you do anything with database you have to have a connection object.

A Connection object represents a connection with a database. When we connect to a database by using connection method, we create a Connection Object, which represents the connection to the database. An application may have one or more than one connections with a single database or many connections with the different databases also.

There are many ways we can establish the connection. The connection object is obtained by the DriverManager.getConnectyion method by supplying the Database location and authentication credentials.

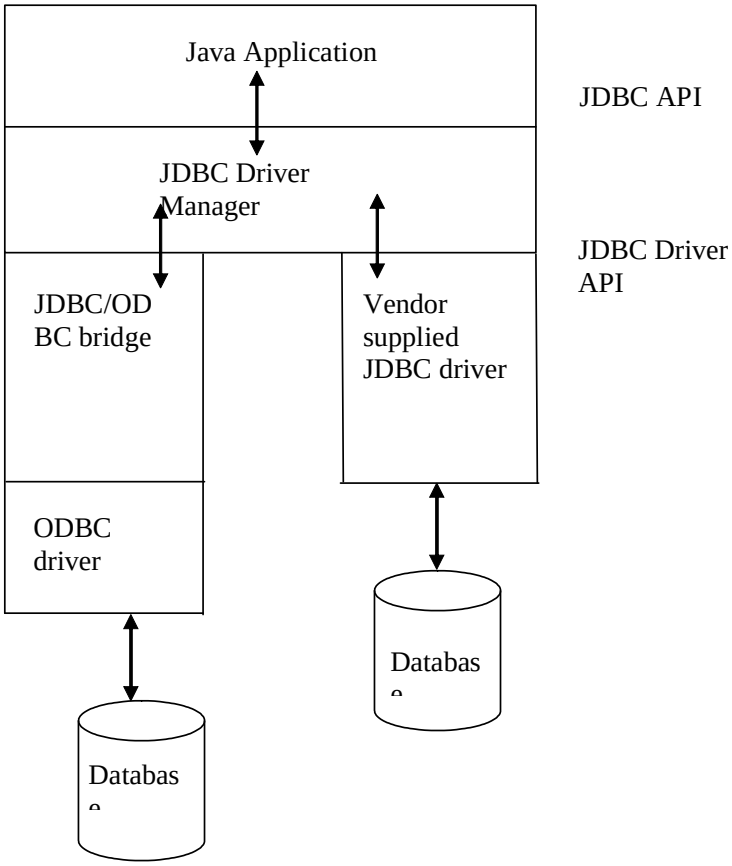
The following are the ways to obtain a Connection object of the database:-

1. DriverManager.getConnection(String URL)
2 DriveManager.getConnection(String URL,String Username, String Password)
3. DriverManager.getConnection (String URL, java.util.Properties props)
4. Driver.connect(String URL, java.util.Properties props)

10.7 JDBC DESIGN CONSIDERATIONS

Starting at 1995, the developers of the Java Technology at Sun start working on extending the standard java library to deal with SQL access to database. But there is simply too many databases on the market like Microsoft Access, Oracle, MySql, PostgreSql, using too many protocols. So it is necessary to provide one standard network protocol for database access. So SUN provide a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors provide their own drivers to plug in to the driver manager. There is a simple mechanism for registering third-party drivers with the driver manager.

So two interfaces were created JDBC API and JDBC Driver API. Application programmers use the JDBC API and database vendors use the JDBC Driver API.



CHECK YOUR PROGRESS -1

1. JDBC is a programming interface between..... and database management system..
2. A program uses a Java class known as ..... to connect to a database.
3. ..... is an interface implemented by a software program to enable interaction with other software
4. The primary function of the driver is to connect to a .....and return a connection object.
5. Type 1 drivers are not.....
6. A .....is the session between your java program and database
7. When we connect to a database by using connection method, we create a..... Object, which represents the connection to the database

10.8 TWO TIER AND THREE TIER CLIENT SERVER MODAL

Let us suppose we are going to write a piece of software that students of a school can use to find out what their current grade is in all their classes. We can structure the program so that a database of grades resides on the server, and the application resides on the client.

When the student wants to know his grades, he manipulates the program by clicking buttons, menu options, etc. The program fires off a query to the database, and the database responds with all the student's grades. Now our application uses all these data to calculate the student's grade, and displays it for him.

This is an example of two-tier architecture. Another example of a two-tier architecture implementation is a Java user interface (Swing/AWT) and batch data processing.

The two tiers are:

1. *Data server* : the database serves up data based on SQL queries submitted by the application.
2. *Client application*: the application on the client computer consumes the data and presents it in a readable format to the student.

This architecture is fine, if we have got the case of a school with 50 students. But suppose the school has 10,000 students. In those case, problem arises. Because every time a student queries the client application, the data server has to serve up large queries for the client application to manipulate.

For this, we create a three-tier architecture by inserting another program at the server level. We call this the server application. Now the client application no longer directly queries the database; it queries the server application, which in turn queries the data server.

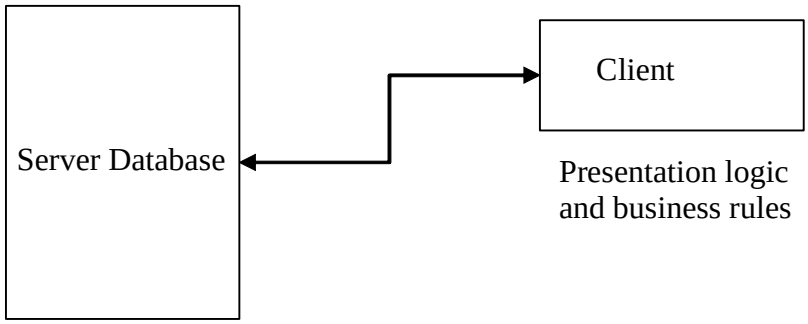


Fig.10.3: Two- Tier Architecture

Now when the student wants to know his final grade, the following steps would occur:

1. The student asks the client application.
2. The client application asks the server application.
3. The server application queries the data server.
4. The data server serves up a record set with all the student's grades.
5. The server application does all the calculations to determine the grade.
6. The server application serves up the final grade to the client application.
7. The client application displays the final grade for the student.

It is a lengthy process on paper, but in reality it is much faster. In Step 6, we can notice that instead of serving up an entire record set of grades, which has to be passed over a network, the server application is serving up a single number, which is a tiny amount of network traffic in comparison.

There are other advantages to the 3-tier architecture, but that at least gives us a general idea of how it works.

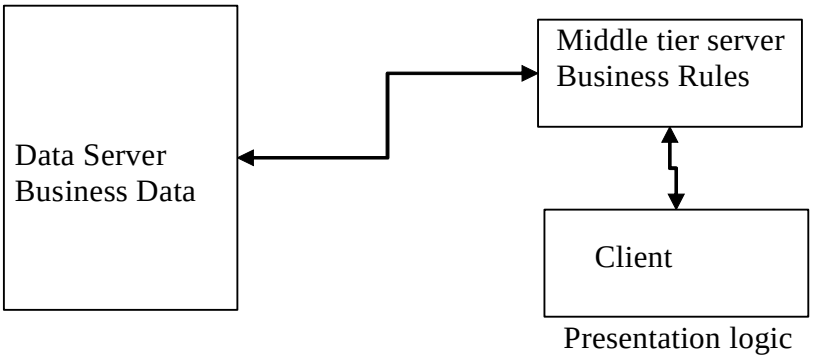
The three-tiers are

Tier 1: the client contains the presentation logic, including simple control and user input validation. This application is also known as a *thin client*.

Tier 2: the middle tier is also known as the *application server*, which provides the business processes logic and the data access.

Tier 3: the data server provides the business data.

An example of 3 tier architecture is *online banking website*. The client application is the web browser we use. The server application is the code written in ASP or JSP or PHP which queries the database (the third tier) for the question-and-answer we request.



2 tier Advantages

- Less Expensive

2 tier Disadvantages

- One can only connects a limited number of users to a server before Database Server spends more time managing connections than processing requests
- it is not scalable, because each client requires its own database session.

3-Tier Advantages

- Improved Security Since the client doesnot have direct access to the database, Data layer is more secure.
- Business Logic is generally more secure since it is placed on a secured central server.
- It is easier to modify or replace any tier without affecting the other tiers.
- Separating the application and database functionality means better load balancing.

3-Tier disadvantages

- Increased Complexity / Effort In General 3-tier Architecture is more complex to build compared to 2-tier Architecture.

10.9 UNDERSTANDING DATA SOURCE

The JDBC API provides the DataSource interface as an alternative to the DriverManager for establishing the connection. A DataSource object is the representation of database or the data source in the Java programming language. DataSouce object is mostly preferred over the DriverManager for establishing a connection to the database.

DataSource has a set of properties that identify and describe the real world data source that it represents. The properties include information about the location of the database server, the network protocol use to communicate with the server the name of the database and so on.

DataSource object works with JNDI (Java Naming and Directory interface) naming service so application can use the JNDI API to access the DataSource object.

In short we can say that the DataSource interface is implemented to provide three kinds of connections:

1. **Basic DataSource class**
   This class is provided by the driver vendor. It is used for portability and easy maintenance.

2. **To provide connection pooling**
   It is provided by the application server vendor or driver vendor. It works with ConnectionPoolDataSource class provided by a driver vendor. Its advantage is portability, easy maintenance and increased performance.

3. **To provide distributed transactions**
   This class works with an XADataSource class, which is provided by the driver vendor. Its advantages are easy maintenance, portability and ability to participate in distributed transactions.

10.10 RESULTSET

ResultSet is a java object that is used for database connectivity to hold the data returned by a select query.

When we run a select query it returns us the data in a table format with each row representing one logical group of data with a number of columns. The result set would contain this table of data and each row can be accessed one by one. We can use the resultset.get() methods to get the data from it. The number of rows returned in a result set can be zero or more. A user can access the data in a result set using a cursor one row at a time from top to bottom. A cursor can be thought of as a pointer to the rows of the result set that has the ability to keep track of which row is currently being accessed.

Types of Result Sets

The ResultSet interface provides methods for retrieving and manipulating the results of executed queries, and ResultSet objects can have different functionality and characteristics. These characteristics are result set type, result set concurrency, and cursor holdability.

The type of a ResultSet object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the ResultSet object.

The sensitivity of the ResultSet object is determined by one of three different ResultSet types:

TYPE_FORWARD_ONLY — the result set is not scrollable i.e. the cursor moves only forward, from before the first row to after the last row.

TYPE_SCROLL_INSENSITIVE — the result set is scrollable; its cursor can move both forward and backward relative to the current position,and it can move to an absolute position.

TYPE_SCROLL_SENSITIVE — the result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.

Result Set Methods

When a ResultSet object is first created, the cursor is positioned before the first row. To move the cursor, you can use the following methods:

next() - moves the cursor forward one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned after the last row.

previous() - moves the cursor backwards one row. Returns true if the cursor is now positioned on a row and false if the cursor is positioned before the first row.

first() - moves the cursor to the first row in the ResultSet object. Returns true if the cursor is now positioned on the first row and false if the ResultSet object does not contain any rows.

last() - moves the cursor to the last row in the ResultSet object. Returns true if the cursor is now positioned on the last row and false if the ResultSet object does not contain any rows.

beforeFirst() - positions the cursor at the start of the ResultSet object, before the first row. If the ResultSet object does not contain any rows, this method has no effect.

afterLast() - positions the cursor at the end of the ResultSet object, after the last row. If the ResultSet object does not contain any rows, this method has no effect.

relative(int rows) - moves the cursor relative to its current position.
absolute(int n) - positions the cursor on the n-th row of the ResultSet object.

10.11 PRAPERED STATEMENTS AND COLLABLE STATEMENTS

Prepared statements:

Java JDBC Prepared statements are pre-compiled SQL statements. Precompiled SQL is useful if the same SQL is to be executed repeatedly, for example, in a loop. **Prepared statements in java** only save our time if we expect to execute the same SQL over again. Every java SQL prepared statement is compiled at some point.

To use a java prepared statement, we must first create a object by calling the Connection.prepareStatement() method. JDBC PreparedStatements are useful especially in situations where we can use a lengthy for loop or while loop to set a parameter to a succession of values. If we want to execute a Statement object many times, it normally reduces execution time to use a PreparedStatement object instead.

The syntax is straightforward: just we have to insert question marks for any parameters that we will be substituting before we send the SQL to the database. As with CallableStatements, we need to call close() to make sure database resources are freed as soon as possible.

An important feature of a PreparedStatement object is that, unlike a Statement object, it is given an SQL statement when it is created. This SQL statement is sent to the DBMS right away, where it is compiled. As a result, the PreparedStatement object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first.

Using Prepared Statements in JDBC, objects can be used for SQL statements with no parameters, we probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that we can use the same statement and supply it with different values each time we execute it.

Callable Statement:

The CallableStatement interface extends PreparedStatement and provides support for output and input/output parameters. The CallableStatement interface also has support for input parameters that is provided by the PreparedStatement interface.

The CallableStatement interface allows the use of SQL statements to call stored procedures. Stored procedures are programs that have a database interface. These programs possess the following:

- They can have input and output parameters, or parameters that are both input and output.
- They can have a return value.
- They have the ability to return multiple ResultSets.

Conceptually in JDBC, a stored procedure call is a single call to the database, but the program associated with the stored procedure may process hundreds of database requests. The stored procedure program may also perform a number of other programmatic tasks not typically done with SQL statements.

Because CallableStatements follow the PreparedStatement model of decoupling the preparation and processing phases, they have the potential for optimized reuse. Since SQL statements of a stored procedure are bound into a program, they are processed as static SQL and further performance benefits can be gained that way. Encapsulating a lot of database work in a single, reusable database call is an example of using stored procedures optimally. Only this call goes over the network to the other system, but the request can accomplish a lot of work on the remote system.

10.12 RESULTSET METADATA OBJECT

A ResultSetMetaData object provides information about the columns in a particular ResultSet instance. One method in the ResultSetMetaData interface returns the number of columns in the ResultSet object as a whole, and the rest of the methods return information about a particular column in the ResultSet object. The column information includes the name of the column, what data type the column can hold, and everything from whether the column value is writable to whether the column value can be used as a search criterion in a WHERE clause. Note that because a RowSet object is derived from a ResultSet object, any of the ResultSetMetaData methods for getting information about a column in a ResultSet object can also be used to get information about a column in a RowSet object.

CHECK YOUR PROGRESS -2

Fill in the blanks :

1. The database serves data based on ..... submitted by the application.
2. The middle tier is also known as the .....server
3. the data server provides the .....
4. The ..... provides the DataSource interface as an alternative to the DriverManager for establishing the connection.
5. ..... is a java object that is used for database connectivity to hold the data returned by a select query.
6. Java JDBC .....statements are pre-compiled SQL statements.
7. The .....interface allows the use of SQL statements to call stored procedures

10.13 LET US SUM UP

- Using JDBC it is possible to communicate with a wide variety of database management system using SQL.
- A program uses a Java class known as JDBC driver to connect to a database.
- The main advantage of JDBC is it provides a standard interface to all database management system.
- An application programming interface (API) is an interface implemented by a software program to enable interaction with other software
- The JDBC API is a Java API for accessing virtually any kind of tabular data.
- The primary function of the driver is to connect to a databse and return a connection object.

- A Connection is the session between your java program and database.
- The JDBC API provides the DataSource interface as an alternative to the DriverManager for establishing the connection
- ResultSet is a java object that is used for database connectivity to hold the data returned by a select query.
- Java JDBC Prepared statements are pre-compiled SQL statements.

10.14 ANSWER TO CHECK YOUR PROGRESS

CHECK YOUR PROGRESS - 1

1. Java programs, 2. JDBC driver, 3. API 4.Database
5. Portable, 6. Connection 7.Connection

CHECK YOUR PROGRESS - 2

1. SQL queries 2. Application 3.data 4. JDBC API
5. ResultSet 6. Prepared 7. CallableStatement

10.15 FURTHER READINGS

1. *"Java- How to Program"*, Pearson Education.
2. "*The Complete Reference -Java 2* " by Herb Hchildt, McGraw-Hill
3. "*JAVA How to Program*", Deitel & Deital, PHI Publication

10.16 POSSIBLE QUESTIONS

1. Write down the four basic steps to JDBC.
2. What are the different types of JDBC driver? Briefly discuss each of them.
3. Discuss 2-tier and 3-tier client server model.
4. What are the advantage and disadvantage of 2-tier and 3-tier client server modal?
5. What are the main difference between prepared statement and collable statement?