ERODE ARTS AND SCIENCE COLLEGE (AUTONOMOUS)

Department of Computer Science (SF)

ODD SEMESTER (2020-2021)

Staff Name : M.DIVYA, M.Sc., M.Phil., **Class : II B.Sc (CS)**

SUBJECT NAME: Programming in JAVA

UNIT I: Overview of Java

Evolution: Java History – Java Features – How Java differs from C and C++ - Java and Internet – Java and World Wide Web - Web Browsers – Hardware and Software Requirements – Java Support Systems – Java Environment. Overview of Java Language: Introduction - Simple Java Program - An Application with Two Class - Java Program Structure -Java Token - Java Statement - Implementing A Java Program - Java Virtual Machine – Command Line Arguments - Programming Style.

UNIT II: Constant, Variables and Data types

Constant - Variables - Data types - Declaration of Variables - Scope of Variables - Typecasting - Standard Default Values. Operators and Expressions: Arithmetic Operator – Relational Operator – Logical Operator – Assignment Operator – Increment Operator and Decrement Operator – Conditional Operator – Bitwise Operator- Special Operator – Arithmetic Expressions – Evaluation of Expression – Precedence of Arithmetic Operators – Type Conversions in expressions – Operator Precedence and Associatively- Mathematical Functions.

UNIT III: Decision Making and Branching, Looping and Classes

Decision Making with If Statement – Simple If Statement - If- Else Statement, Nesting of If-Else Statement - Else If Statement - Switch -? Operator. Looping: While, Do-While, for, Comma Statements – Continue Classes: - Creating Object, Accessing Class Members - Constructors - Methods of Overloading - Static Members - Nesting of Methods- Inheritance - Overriding Methods - Final Variables andMethods - Final Classes - Finalizer Methods - Abstract Methodsand Classes - Visibility Control.Arrays, Strings and Vectors: Creating an Array - Two Dimensional Arrays - Strings - Vectors - Wrapper Classes.

UNIT IV: Interfaces, Packages of Multithreaded Programming

Interfaces: Multiple Inheritance - Defining Interfaces -Extending Interfaces - Implementing Interfaces - Accessing Interfaces Variable. Packages: Using System Packages - Naming Conventions - Creating Packages - Accessing A Packages - Adding A Class to A Packages -Hiding Classes. Multithreaded Programming: Creating Threads - Extending the Thread Class -Stopping and Blocking A Thread - Life Cycle of a Thread - Using Thread Methods - Thread Exceptions - Thread Priority.

UNIT V: Managing Errors, Exceptions and Applet Programming

Managing Errors: Introduction - Types of Errors. Exception: Syntax Of Exception Handling Code - Multiple Catch Statement - Using Finally Statement - Throwing Our Own Exceptions - Using Exception For Debugging Applet Programming: Local And Remote Applets - Preparing To Write Applets - Building Applet Code - Applet Life Cycle - Creating And Executable Applet - Designing Web Page - Applet Tag - Adding Applet To Html File - Running The Applet - More About Applet Tag – Passing Parameters To Applet - Aligning the Display - More About Html Tags - Displaying Numerical Values. Graphics Programming: Graphics Class- Lines and rectangle – Circles and Ellipses – Drawing Arcs – Drawing Polygons – Line Graphs - Drawing Bar Charts.

TEXT BOOK:

E. Balagurusamy, “Programming with JAVA”, Tata McGraw Hill Publishing Company, Third Edition, 2010.

(UNIT I: CHAPTER 2,3 UNIT II: CHAPTER 4,5 UNIT III: CHAPTER 6,7,8,9 UNIT IV: CHAPTER 10,11,12 UNIT V: CHAPTER 13,14,15)

REFERENCE BOOKS:

- C. Xavier, “Programming with JAVA 2”, Scitech Publications (India) Pvt. Ltd., Eighth Edition, 2009.
- S.Sagayaraj, R.Denis, P.Karthik, D.Gajalakshmi, “ Java Programing for Core and Advanced Learners”, Universities Press(India) Pvt., 2018.
- P.Radhakrishnan, “ Object Oriented Programming Through JAVA”,Universities Press(India) Pvt., 2007.

UNIT I: Overview of Java

Introduction

- Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991.
- Originally called **"Oak"** by James Gosling, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices
- Like TVs, VCRs, toasters and such other electronic machines.
- This goal had a strong impact on the development team to make the language simple, portable and highly reliable.
- Java is a really simple, reliable, portable, and Powerful language. Some important milestones in the development of Java.

JAVA History

- **1990** - **Sun Microsystems** decided to develop special software that could be used to manipulate consumer electronic devices.
    - A team of Sun Microsystems programmers headed by **James Gosling**.
- 1991 - Most popular object-oriented language in C++, the team announced a new language named as **"Oak".**
- 1992 - New language to control a list of home appliances using a hand-held device with a **tiny touch sensitive screen**.
- 1993 - The World Wide Web CNWWJ appeared on the Internet and transformed the **text-based Internet** into a **graphical-rich environment**.
- 1994 - The team developed a Web browser called "**HotJava**" to locate and run applet programs on Internet.
- 1995 - Oak was renamed **"Java",** due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
- 1996 - Object-oriented programming language **Java** found its home.

Java Feature

Java language to be not only reliable, portable and distributed but also simple, compact and interactive. Sun Microsystems officially describes Java with the following attributes:

- Compiled and Interpreted
- Platform-Independent and Portable
- Object-Oriented
- Robust and Secure
- Distributed
- Familiar, Simple. and Small
- Multithreaded and Interactive
- High Performance
- Dynamic and Extensible

Compiled and Interpreted

- A computer language is either compiled or interpreted. Java combines both these approaches thus making Java a two-stage system.
- First stage, Java **compiler** translates source code into byte code instructions.
- Second stage, Java **interpreter** generates byte code that can be directly executed by the machine that is running the Java program.

Platform-Independent and Portable

- Java over other languages is its **portability**. Java programs can be easily moved from one computer system to another, anywhere and anytime.
- Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs.
- A **Java applet** from a remote computer onto our local system via Internet and execute it locally.
- Java programs can execute in different operating system. (Windows, Linux, Unix, DOS)

Object-Oriented

- Java is a true object-oriented language. Almost everything in Java is an object. All program code and data reside within objects and classes.
- Java comes is extensive set of classes, arranged in packages, that we can use in our programs by inheritance.

Robust and Secure

- Java is a **robust** language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types.
- Java also incorporates the concept of exception handling which captures series errors and eliminates any risk crashing the system.
- **Security** becomes an important issue for a language that is used for programming on Internet. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet.
- The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

Distributed

- Java applications can open and access remote objects on Internet as easily as they can do in a local system.
- This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

Simple, Small and Familiar

- Java is a small and simple language. Many features of C and C+ + that are either redundant or sources of unreliable code are not part of Java.
- **For example**, Java does not use pointer, preprocessor header files, goto statement and many others. It also eliminates· operator overloading and multiple inheritance.

Multithreaded and Interactive

- Multithreaded means handling multiple tasks simultaneously.
- Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another.

High Performance

- Java performance is impressive for an interpreted language.
- Mainly due to the use of intermediate bytecode. According to Sun, Java speed is comparable to the native C/C++.

Dynamic and Extensible

- Java is a dynamic language. Java is operable of dynamically linking in new Class libraries, methods, and objects.
- Java is also determine the type of class through a query, making it possible to either dynamically link.

How differs from C and C++

Java and C

Java and C Java is a lot like C but the major difference between Java and C is that Java is an object-oriented language and has mechanism to define classes and objects.

- Java does riot include the C unique statement keywords **goto, size of, and typedef.**
- Java does not contain the data types **struct, union and enum .**
- Java does not define the type modifiers **keywords auto, extern, register, signed, and unsigned.**
- Java does not support an explicit **pointer** type.
- Java does not have a preprocessor and therefore we cannot use **# define, # include, and # ifdef** statements.
- Java adds new operators such as instance of and> > >**.**
- Java adds labelled **break and continue** statements.
- Java adds many features required for **object-oriented programming.**

Java and C++

Java is a true object-oriented language while c++ is basicallyC with object-oriented extension. Listed below are some major C+ + features that were intentionally omitted from Java or .. significantly modified .

- Java does not support **operator overloading**.
- Java does not have **template** classes as in C++ .
- Java does not support **multiple inheritance** of classes. This is accomplished using a new feature called "interface".
- Java does not support **global variables**. Every variable and method is declared within a class and forms part' of that class.
- Java does not use **pointers**.
- Java has replaced the destructor function with a **finalize**() function.
- There are no **header files** in Java.

Java and Internet

- Java is strongly associated with the Internet because of the fact that the first application program written in Java was Hot Java, a Web browser to run applets on Internet.
- Internet users can use Java to create applet programs and run them locally using a "Java-enabled browser" such as Hot Java.

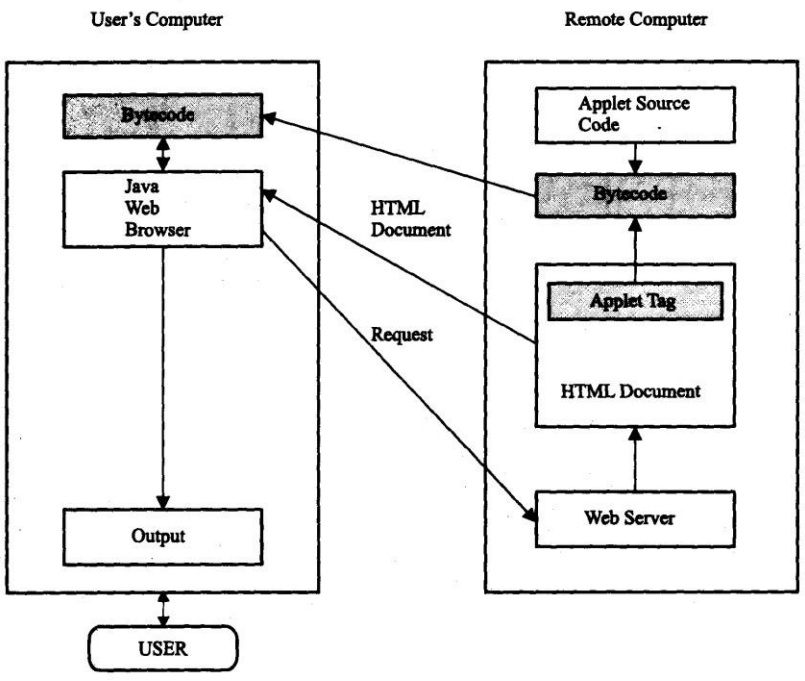
Java and World Wide Web

- World Wide Web (WWW) is an open-ended information retrieval system designed to be used in the Internet's distributed environment.
- This system contains what are known as Web pages that provide both information and controls.

Web browser

- A large portion of the Internet is organized as the World Wide Web which uses hypertext. Web browsers are used to navigate through the information found on the net.
- They allow us to retrieve the information spread across the Internet and display it using the hypertext markup language (HTML).

Fig. 2.4



Java's interaction with the web

Hardware and software requirements

Java is currently supported by windows, sun solar system and Macintosh and UNIX machines.

Java Support Systems.

Java and java enabled web browser on the internet requires a verity systems.

- Internet connection
- Web browser
- HTML
- Applet Tag
- Java Code
- Byte Code

Java Environment

- Java environment includes large number of Tools and hundreds of classes and Methods.
- The development of tools Part of a system known as Java Development Kit (JDK) and Classes and method is a part of Java Standard Library (JSL).

Java Development Kit (JDK)

The java Development Kit is Collection of Tools its used for developing running a java Programs.

It includes

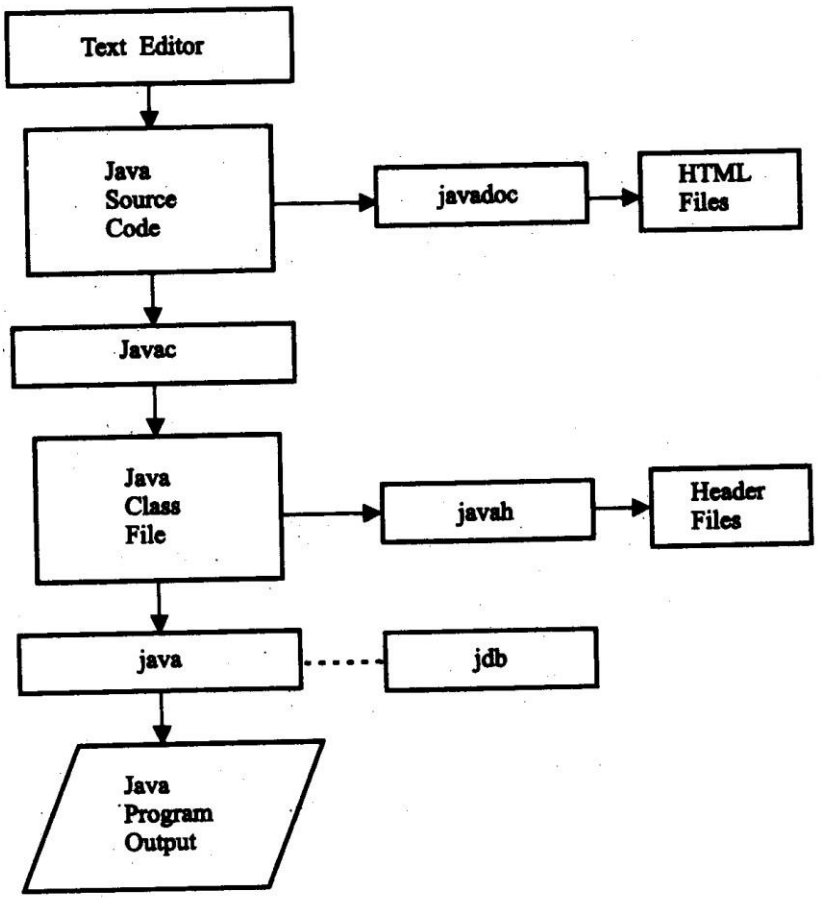
- Applet viewer (Viewing applet)
- Javac (Compiler)
- Java (interapreter)
- Javap (Disassembler)
- Javah (C Header Files)
- Jdb (Debugger)

Java Standard Library

The,Java Standard Library includes hundreds of classes and methods grouped into six functional packages .

- **Language Support Package**: A collection of classes and methods required for implementing basic features of Java .
- **Utilities Package**: A collection of classes to provide utility functions such as date and time functions.
- **Input/output Package**: Acollection of classes required for input/output manipulation.
- **Networking Package**: A collection of classes for communicating with other computers via Internet.
- **AWT Package**: The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.
- **Applet Package**: This includes a set of classes that allows us to create Java applets.

Fig. 2.5



Process of building and running Java application programs

Overview of java Language

Introduction

Java is a general-purpose, object-oriented programming language. We can develop two types of

Java programs:

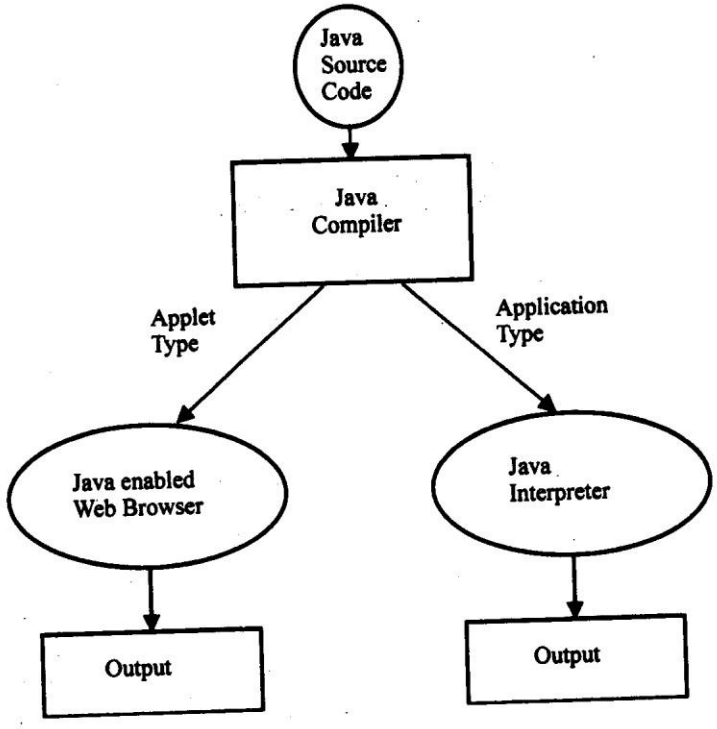
- Stand-alone applications
- Web applets

Executing a stand-alone Java program involves two steps:

1. Compiling source code into bytecode using **Javac compiler**

2. Executing the byte code program using **Java interpreter.**

Fig. 3.1



Two ways of using Java

- An applet located on a distant computer (Server} can be downloaded via Internet and executed on a local computer (Client) using a Java-capable browser.
- We can develop applets for doing everything from simple animated graphics to complex games and utilities

SIMPLE JAVA PROGRAM

Program 1:

```
class SarnpleOne
{
  public static void rnain(String args[ ])
  {
    Systern.out.println("Java is better than C++.");
  }
}
```

Output: **Java is better than C++**

Program 2:

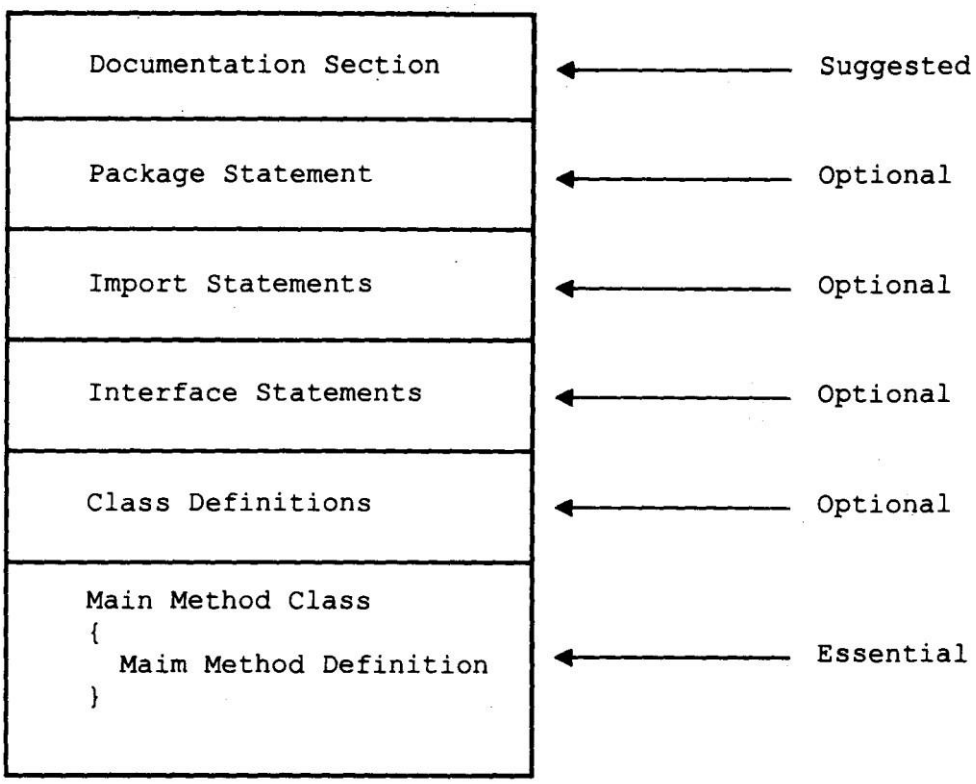
```
class Hall
{
        float length;
        float breadth;
        void getdata(float a, float b)
        {
                length = a·;
                breadth = bi
        }
}
Class  HallArea
{
        public static void main(String args[ ])
        {
        float area;
        HallArea h1 = new HallArea( ); // Creates an object room1
        h1.getdata (14, 10);
        area = h1.length * h1. breadth;
        System.out.println("Area =" + area);
        }
}
```

Output: **Area = 140**

Java Program Structure

- A Java program may contain many classes of which only one class defines a main method.
- Classes contain data members and methods that operate on the data members of the class.
- Methods may contain data type declarations and executable statements.

Fig. 3.2



General structure of a Java program

Documentation Section

The documentation section comprises a set of comment lines giving the

- Name of the program,
- Author Name and
- Other details, of the programmer.
- Comments must explain why and what of classes and how of algorithms

Types of Java Comments

- Single Line Comment -
    - The single line comment is used to comment only one line.

```
//Here, i is a variable
```

- Multi Line Comment
    - The multi line comment is used to comment multiple lines of code.

```
/* Let's declare and
 print variable in java. */
```

- Documentation Comment
    - the doc umentation comment is used to create documentation API. To create documentation AP, you need to use javadoc.tool

```
/** The Calculator class provides methods to get addition and subtr
action of given 2 numbers.*/
```

Package Statement

- The first statement allowed in a Java file is a package statement.
- This statement declares a package name and informs the compiler that the classes defined here belong to this package.

Example: package student; The package statement is optional.

Import Statements

This is similar to the # include statement in C++.

Example: import student. test; This statement instructs the interpreter to load the test - class obtained in the package student.

Using import statements, we can have access to classes that are part of other named packages.

Example:

import java.io.*;

import java.lang.*;

Interface Statements

- An interface is like a class but includes a group of method declarations.
- This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.
- Interface is a new concept in Java.

Class Definitions

- A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program.
- These classes are used to map the objects of real-world problems. The number of classes used depends on the complexity of the problem.

Main Method Class

- Every Java stand-alone program requires a main method as its starting point, this class is the essential part of a Java program.
- A simple Java program may contain only this part. The main method creates objects of various classes and establishes communications between them.
- On reaching the end of main, the program terminates and the control passes back to the operating system.

JAVA TOKENS

- Java Tokens are the smallest individual building block or smallest unit of a Java program; the Java compiler uses it for constructing expressions and statements.
- Java program is a collection of different types of tokens, comments, and white spaces.
- There are various tokens used in Java:

Java language includes five types of tokens.

They are:

- Reserved Keywords
- Identifiers
- Literals
- Operators
- Separators

JAVA CHARACTER SET

- The smallest units of Java language are the characters used to write Java tokens.
- Bese characters are defined by the Unicode character set, an emerging standard that tries to create characters for a large number of scripts worldwide.

KEYWORDS

- Keywords are an essential part of a language definition. They implement specific features of the language.
- Java language has reserved 50 words as keywords. Table 3.1 lists these keywords. These keywords, combined with operators and separators according to syntax, form definition of the Java language.
- Understanding the meanings of all these words is important for Java programmers.

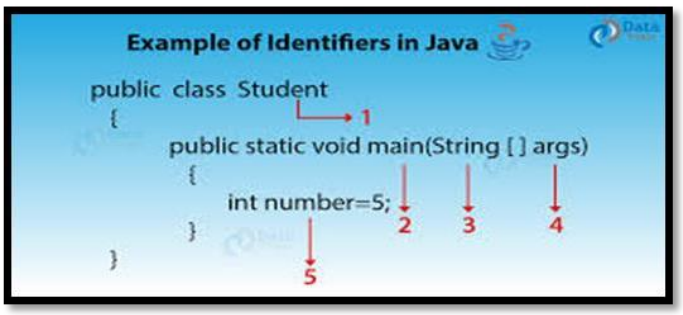
abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while	true	False and Null

IDENTIFIERS

Identifiers are programmer-designed tokens. They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program.

Java identifiers follow the following rules:

1. They can have alphabets, digits, and the underscore and dollar sign characters.
2. They must not begin with a digit.
3. Uppercase and lowercase letters are distinct.
4. They can be of any length.



Identifier must be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily read. Java developers have followed some naming conventions.

LITERALS

Literals in Java are a sequence of characters (digits, letters, and other characters) that represent constant values to be stored in variables. Java language specifies five major typeSof literals.

They are:

- Integer literals
- Floating int literals
- Character literals
- String literals
- Boolean literals

Each of them has a type associated with it. The type describes how the values behave and how they are stored.

JAVA OPERATORS

Java operators are symbols that are used to perform mathematical or logical manipulations. Java is rich with built-in operators.

There are many types of operators available in Java such as:

- Arithmetic Operators
- Unary Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Compound Assignment Operators
- Conditional Operator
- instanceof Operator
- Member Selection or Dot Operator

SEPARATORS

Separators are symbols used to indicate where groups of code are divided and arranged. They basically define the shape and function of our code.

Name	What it is used for
parentheses( )	Used to enclose parameters in method definition and invocation, also used for defining precedence in expressions, containing expressions for now control, and surrounding cast types

- ***Examples:***
  - System.out.println("addition of a and b is:"+c);
  - getdata (int a , int b);
  - public static void main(String args[ ])
  - // uses paranthesis or ()

braces{ } Used to contain the values of automatically initialized arrays and to define a block of code for classes, methods and local scopes.

- ***Example:***

```
class Students
{
   int a,int b;
   getdata();
   display_data();
}
```

brackets [ ] Used to declare array types and for dereferencing array values.

- ***Example:***
  - String name [25];
  - int a[10];

semicolon ; Used to separate statements.

- ***Example:***
  - int a=10,b=15,c;
  - c=a+b;

comma, Used to separate consecutive identifiers in a variable declaration, also use to chain statements together inside a 'for' statement

- ***Example:***
  - int a=10,b=15,c;
  - for(i=0,j=0;i<=10;i++)

period . Used to separate package names from sub-packages and classes; also used to separate a variable or method from a reference variable.

- ***Example:***
  - s2.getdata();
  - s2.display_data();

JAVA STATEMENTS:

- The statements in Java are like sentences in natural languages.
- A statement is an executable. Combination of tokens ending with ·a semicolon (;) mark.
- Statements are usually executed in sequence in the order in which they appear.
- However, it is possible to control the flow of execution, if necessary, using special statements.
- Java implements several types of statements

EMPTY STATEMENT

These do nothing and are used during program development as a place holder

Example:

for(int i = 0; i < 10; a[i++]++) // Increment array elements

/* empty */; // Loop body is empty statement

LABELED STATEMENT

A labeled statement is simply a statement that has been given a name by prepending a identifier and a colon to it. Labels are used by the break and continue statements.

For example:

rowLoop: for(int r = 0; r < rows.length; r++) { // A labeled loop

colLoop: for(int c = 0; c < columns.length; c++) { // Another one

break rowLoop; // Use a label

}

}

EXPRESSION STATEMENT

Most statements are expression statements Java has seven types of Expression statements: Assignment, Pre-Increment, Pre-Decrement, Post-Increment, Post-Decrement, Method Call and Allocation Expression.

Example:

```
a = 1; // Assignment

x *= 2; // Assignment with operation

i++; // Post-increment

--c; // Pre-decrement

System.out.println("statement"); // Method invocation
```

SELECTION STATEMENT

These select one of several control flows. There Java has four types of Expression statements:

- If Statement
- If ..Else statement
- Else ..if Statement
- Switch Statement

ITERATION STATEMENT

These specify how and when looping will take place. There are three types of iteration statements: while, do and for.

JUMP STATEMENT

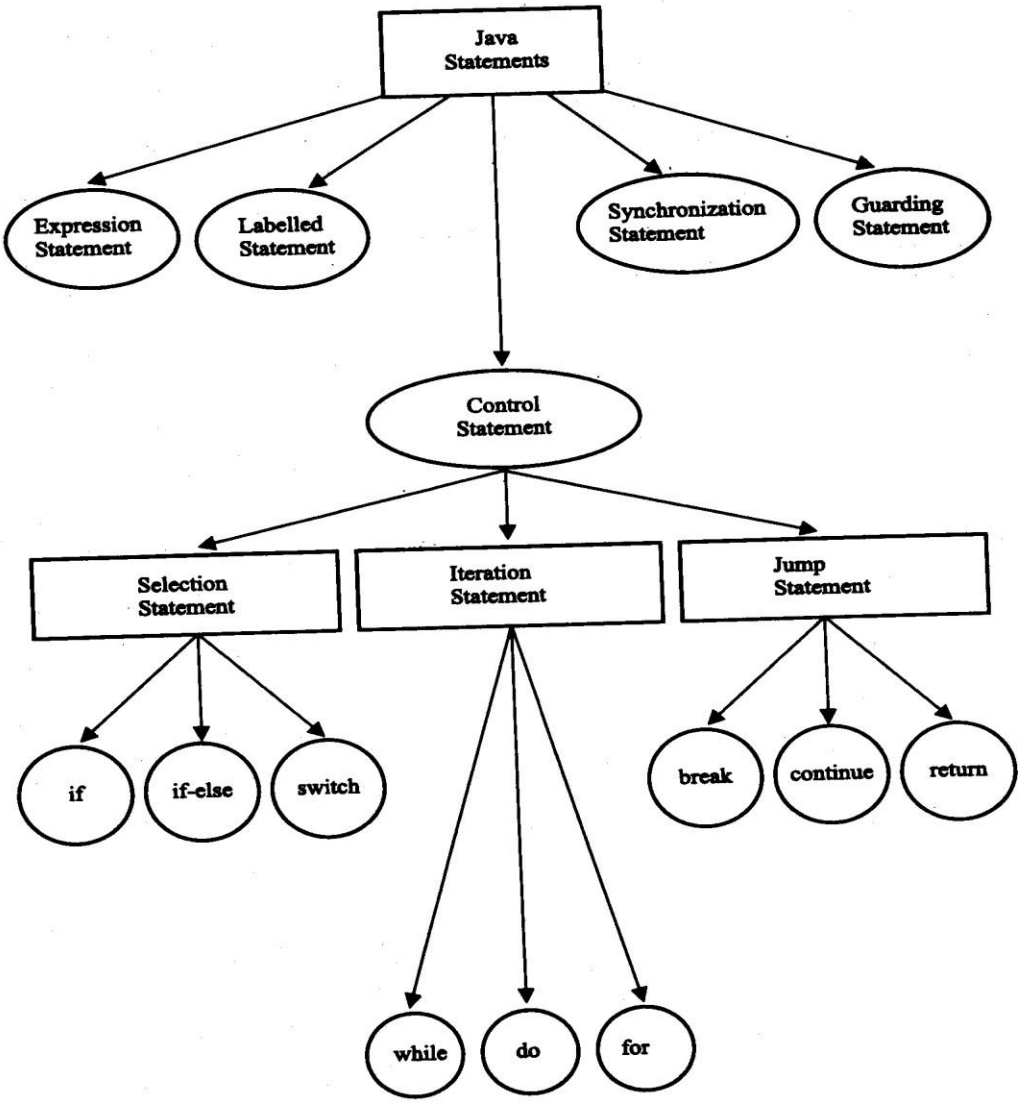
Jump Statements pass control to the beginning or end of the current block, or to a labeled statement. Such labels must be in the same block, and continue labels must be on an iteration statement. The four types of Jump statement are break, continue, return and throw

SYNCHRONIZATION STATEMENT

These are used for handling issues with multithreading.

GUARDING STATEMENT

Guarding statements are used for safe handling of code that may cause exceptions (such as division by zero). These statements use the keywords try, catch, and finally.



Classification of Java statements

Implementing a Java Program

Implementation of a Java application program involves a series of steps.

They include

- Creating the program
- Compiling the program
- Running the program.

Remember that, before we begin creating the program, the Java Development Kit (JDK) must be properly installed on our system.

CREATING THE PROGRAM

We can create a program using any text editor. Assume that we have entered the following

Program:

```
class Test
{
        public static void main (String args[ ] )
        {
        System.out.println(“Hello!");
        System.out.println(“Welcome to the world of Java.");
        System.out.println(“Let us learn Java.");
        }
}
```

We must save this program in a file called **Test.java** ensuring that the filename contains the class name properly.

This file is called the **source.fi1e**.Note that all Java source files will have the extension java. Note also that if a program contains multiple classes, the file name must be the class name of the class containing the main method.

COMPILING THE PROGRAM

To compile the program, we must run the Java Compiler **javac,** with the name of the source file

on the command line as shown below:

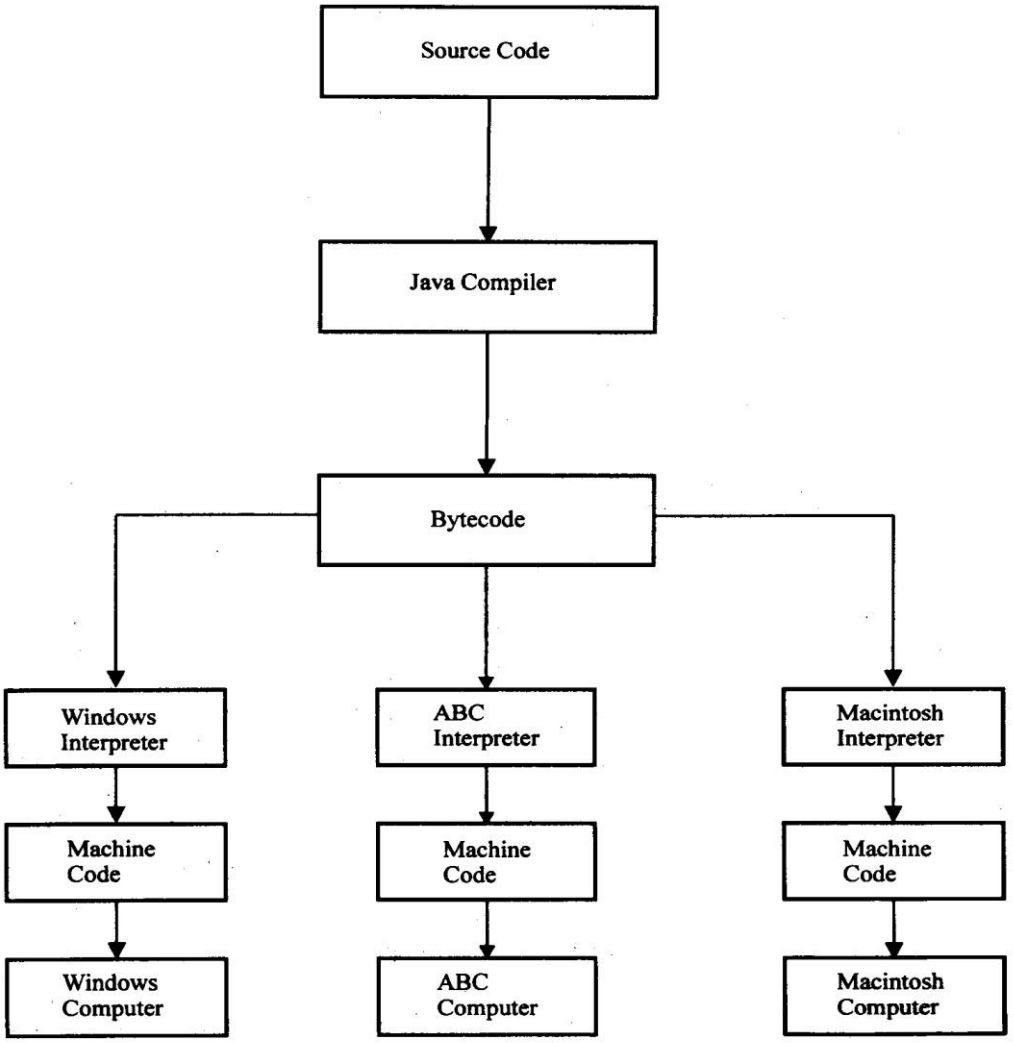
javac Test. java

If everything is OK, the **javac compiler** creates a file called **Test.class** containing the

bytecodes of the program. Note that the compiler automatically names the **bytecode** file as

<classname> .class

Fig. 3.5



Implementation of Java programs

RUNNING THE PROGRAM

We need to use the Java interpreter to fUn a stand-alone program. At the command prompt, type

Java Test

Now, the interpreter looks for the main method in the program and begins execution from there. When executed, our program displays the following:

Hello!

Welcome to the world of Java.

Let us learn Java.

Note that we simply type "Test" at the command line and not "Test. class" or "Test.java".

JAVA VIRTUAL MACHINE

All language compilers translate source code into machine code for a specific computer. Java compiler also does the same thing.

Then, how does Java achieve architecture neutrality? The answer is that the Java compiler produces an intermediate code known as byte code for a machine that does not exist.

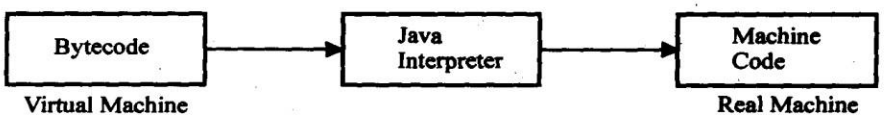
This machine is called the Java Virtual Machine and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer.

Fig. 3.6

Process of compilation

The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine as shown in Fig. 3.7. Remember that the interpreter is different for different machines.

Fig. 3.7



Process of converting bytecode into machine code

COMMAND LINE ARGUMENTS

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution.

This is achieved in Java programs by using what are known as command line arguments.

Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution.

Here, we have not supplied any command line arguments.

Even if we supply arguments, the program does not know what to do with them.

We can write Java programs that can receive and use the arguments provided in the command line. Recall the signature of the main method used in our earlier **example**

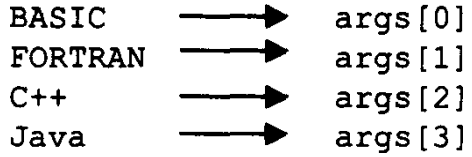
programs:

public static void main (String args[ ])

We can simply access the array elements and use them in the program as we wish. For example, consider the command line

java Test BASIC FORTRAN C++ Java

This command line contains four arguments. These are assigned to the array args as follows:




```
Class ComLineTest
{
        public static void main(String args[ ] )
        {
        int count, i=O;
        String string;
        count = args.length;
        System.out.println("Number of arguments = " + count);
        while (i < count) .
        {
        string = args[i];
        i = i + 1;
        System.out.println(i+ " : " + "Java is " +string+ "!");
        }
        }
}
```

java ComLineTest Simple Object_Oriented Distributed Robust Secure

Portable. Multithreaded Dynamic

The index i is incremented using a while loop until all the arguments are accessed. The number of arguments is obtained by statement

count = args.length;

The output of the program would be as follows:

Number of arguments = 5

1 : Java is Simple!

2 : Java is Object_Oriented!

3 : Java is Distributed!

4 : Java is Robust!

5 : Java is Secure!

Note how the output statement concatenates the strings while printing.

PROGRAMMING STYLE

Java is a free form language. We need not have to indent any lines to make the program work properly. Java system does not care where on the line we begin typing. While this may be a license for bad programming, we should try to use this fact to our advantage for producing readable programs.

```
System
.
Out
.
println
(
Java is Wonderful!"
) ;
```

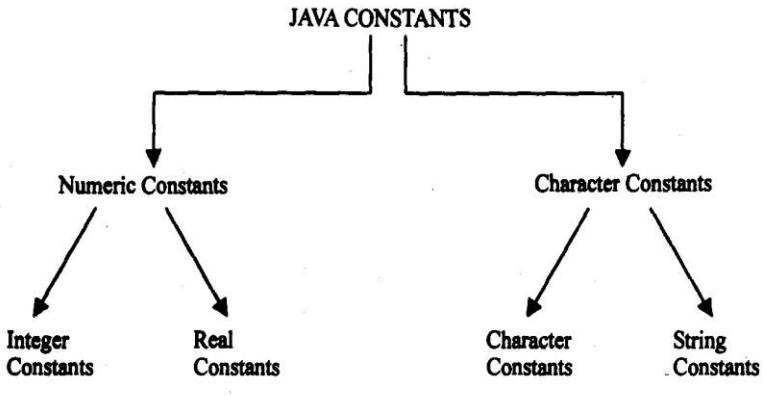
UNIT II: Constant

Java Constant:

- **Constants** in Java refer to fixed values that do not change during the execution of a program. Java supports several types of constants.

48 Programming with Java: A Primer

Fig. 4.1



Java constants

Integer Constant

- An **integer constant** refers to a sequence of digits. There are three types of integers
  - Decimal integer,
  - Octal integer and
  - Hexadecimal integer.

Decimal Integer:

- Decimal integers consist of a set of digits, 0 through 9, preceded ,by an optional minus sign.
- **Example:**
  - 123 , -321, 0, 654321

Octal Integer

- An Octal integer constant consists of any combination of digits from the set 0 through 7.
- **Examples:**
  - 037, 0, 0435, 0551

Hexadecimal Integer

- A sequence of digits preceded by Ox or OX is considered as hexadecimal integer (hex integer).

- **Examples:**
    - OX 2, OX 9F, Ox bcd, Ox

Real Constants

- These numbers are shown in **decimal notation**, having a whole number followed by a decimal point and the fractional part, is an integer is called **Real Numbers** such as Distances, heights, temperatures, prices, and so on.
- **Examples:**
    - 0.0083, -0.75, 435.36

Single Character Constants·

- A Single Character Constant (or simply character constant) contains a single character enclosed within a pair of single quote marks.
- **Examples** of character constants are:
    - '5', 'X', ';', '.'

String Constants

- A String Constant is a sequence of characters enclosed between double quotes.
- The characters may be alphabet, digits, special characters and blank spaces.
- **Examples** are**:**
    - "Hello Java", "1997", "WELLDONE", "?...!", "5+3", "X"

Backslash Character Constants

- Java supports some special backslash character constants that are used in output methods. **For example**, the symbol '\n' stands for newline character.
- A list of such backslash character constants is

Constant'	**Meaning**
'\b'	Back Space
,\f'	Form Feed
'\n'	New Line
, \r'	Carriage Return
'\t'	Horizontal Tab
'\' ,	Single Quote
,\'"	Double Quote
'\\'	Backalash

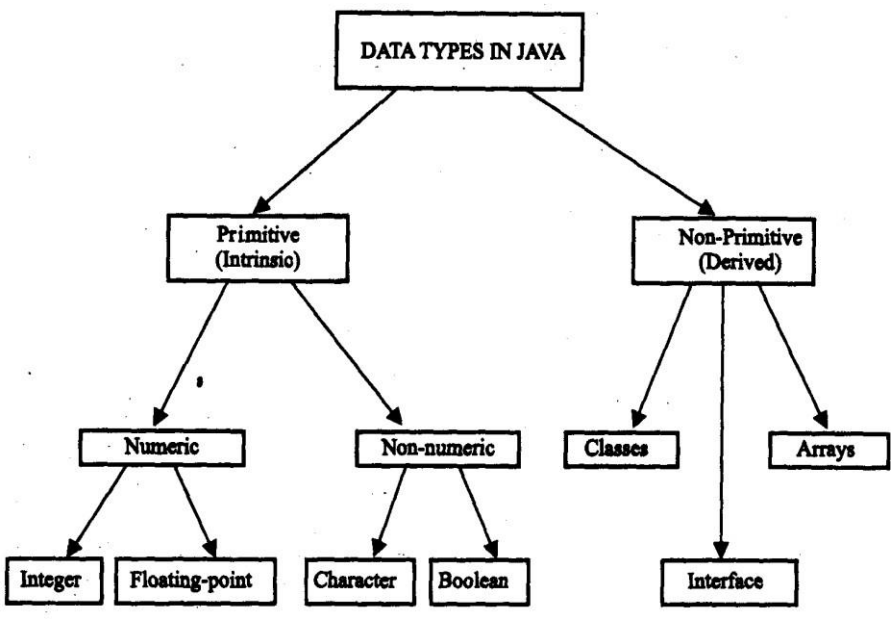
Variables

- A **Variables** is an identifier that denotes a storage location used to store a data value.
- **Constants** that remains unchanged during the execution of a program.
- A **Variable** may take different values at different times during the execution of the program.
- **Examples** of Variable names are:
    - Average
    - height
    - total_height
    - classStrength
- **Variable** names may consist of following conditions:
    - They must not begin with a digit.
    - Uppercase and lowercase are distinct. This means that the variable Total is not the same as total or TOTAL.•
    - It should not be a keyword.
    - White space is not allowed.
    - Variable names can be of any length.

Data Types

- Every variable in Java has a **data type**. Data types specify size and type of values that can be stored. Java language is rich in its data types.

Fig. 4.2



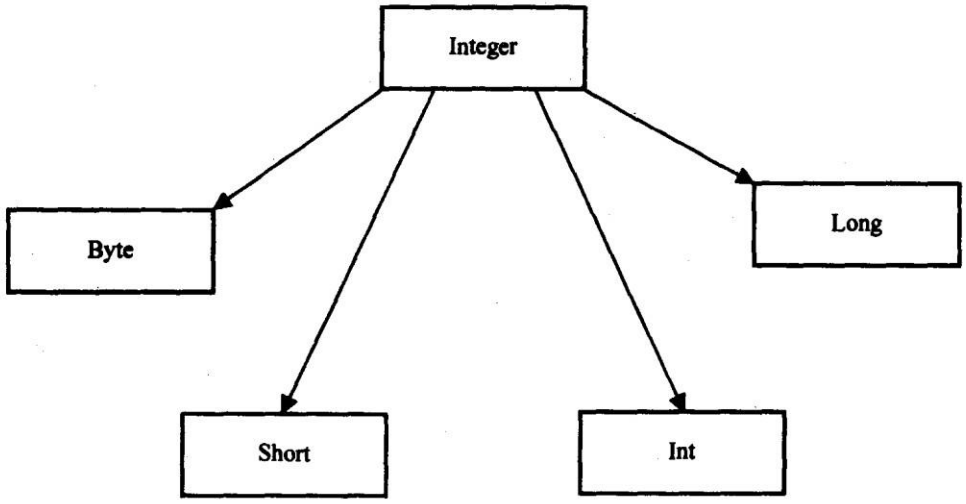
Data types in Java

Integer Types

- **Integer types** can hold whole numbers such as 123, -96, and 5639. The size of the values that can be stored sepends on the integer data type we choose.
- Java supports **four** types of integers

52 Programming with Java: A Primer

Fig. 4.3



Integer data types

Size and Range of Integer Types

Type	Size	Minimum value	Maximum value
Byte	One byte	-128	127
Short	Two bytes	-32, 768	32, 767
Int	Four bytes	-2, 147, 483, 648	2, 147, 483, 647
Long	Eight bytes	-9, 223, 372, 036, 854, 775, 808	9, 223, 372, 036, 854, 775, 807

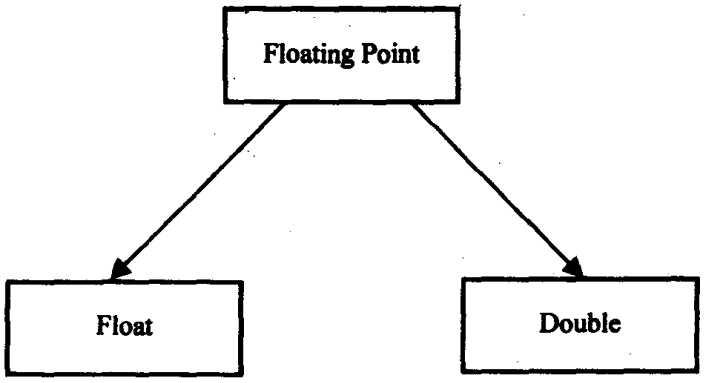
Floating Point Types

Floating point type to hold numbers containing fractional parts such as 27.59 and -1.375 known as floating point numbers.

Size and Range of Floating Point Types

Type	Size	Minimum value	Maximum value
float	4 bytes	3.4e-038	3.4e+038
double	8 bytes	1.7e-308	1.7e+308

Fig. 4.4



Floating point data types

Character Type

- Java provides a character data type called char. The char type assumes a size of 2 bytes.
- Basically, it can hold only a **single character.**

Boolean Type

- Boolean type is used when we want to test a particular condition during the execution of the program.
- There are only two values that a Boolean type can take: **true or false.**

Declaration of Variables

- Variables are the names of storage locations. After designing suitable variable names.
- We Must declare them to the compiler. Declaration does three things:

1. It tells the compiler what the **variable name** is.
2. It specifies what type of **data the variable** will hold.
3. The place of declaration (in the program) decides the **scope of the variable**.

A variable can be store a value of any data type. The general form of declaration of variable is:

```
type variable1, variable2, ........., variableN;
```

Variables are separated by comma; declaration statement must end with a semicolon.

Examples:

int count;

float avg, prize;

double pi;

byte b;

char c1,c2,c3;

Scope of Variables

Java variables are actually classified into three kinds:

- Instance variables,
- Class variables, and
- Local variables.

Instance Variables

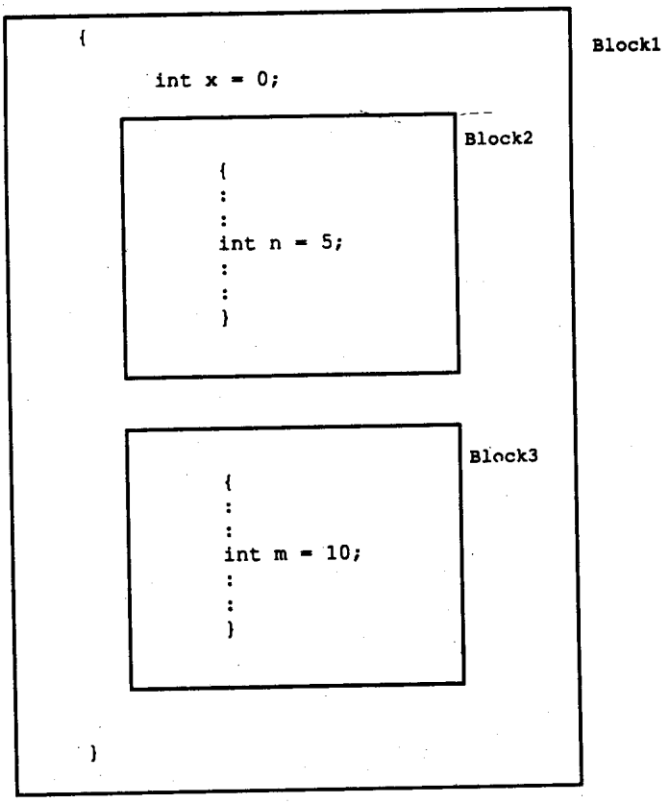
- **Instance and class variables** are declared inside a class.
- Instance variables are created when the objects are instantiated' and therefore they are associated with the objects.
- They take different values for each object

Class Variables

- Class variables are global to a class and belong to the entire set of objects that class creates.

Local Variable

- Variables declared and used inside methods are called **local variables**.
- Local variables can also be declared inside program blocks that are defined between an opening brace { and a closing brace }.



Nested program blocks

Typecasting

- There is a need to store a value of one type into a variable of another type.
- We must cast the value to be stored by proceeding it with the type name in parentheses.
- The syntax is:

type variable l = (type) variable2;

The process of converting one data type to another is called **typecasting**.

Examples:

```
int m = 50;
byte n = (byte)m;
long count = (long)m;
```

Table 4.5 Casts That Result In No Lou of Infonnatlon

From	To
byte	short, char, int, long, float, double
short	int, _J,ong, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	douple

Example Program:

```
class TypeWrap
{
   public static void main (String args[])
  {
     System.out.println("Variables-created");
     char c =.•x";
     byte b = 50;
     shorts = 1996;
     int i = 123456789;
     long l = 1234567654321L;
     float f1 = 3.142F;
     float f2 = 1.2e-SF;
     double d2 = 0.000000987;
     System.out.println("Original Values");
     System.out.println(" c =" + c);
     System.out.println(" b =" + b);
     System.out.println(" s =" + s);
     System.out.println(" i =" + i);
     System.out.println(" l="+l);
     System.out.println(" f1 =" + f1);
     System.out.println(" f2 =" + f2);
     System.out.println.(" d2 =" + d2);
     System.out.println(" ");
```

```
        System.out.println("Types converted");
        short sl = (short)b;
        short s2 = (short) i; // Produces incorrect result
        float n1 = (float) 1;
        int m1 = (int)f1; // Fractional part. is lost
        System.out.println(" (short)b ="+ sl);
        System.out.println(" (short)i ="+ s2);
        System.out.println(" (float) l="+ n1);
        System.out.println(" (int)f1 ="+ m1);
    }
}
```

Output of the Program

```
Original Values
c = x
b = 50
s = 1996
i = 123456789
1 = 1234567654321
f1 = 3.142
f2 = 1.2e-005
92 = 9.87e-007
Types converted
(short)b = 50
(short)i = -13035
(float)l = 1.23457e+012
(int)f1 = 3
```

Standard Default Values

- Every variable has a default value. If we don't initialize a variable when it is first created,
- Java provides default value to that variable type automatically.

Type of variable		Default value
• byte	:	Zero (byte) 0
• Short	:	Zero (short) 0
• int	:	Zero 0
• long	:	Zero OL
• float	:	O.Of
• double	:	O.Od
• char	:	null character
• boolean	:	false
• reference	:	null

OPERATORS

- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- Operators are used in programs to manipulate data and variables.
- Java operators can be classified into a number of related categories as below:
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Increment and decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators

Arithmetic operators

- Java provides all the basic arithmetic operators.

Operator	Meaning
+ - * / %	Addition or unary plus Subtraction or unary minus Multiplication Division Modulo division

Integer Arithmetic

- Both the operands in a single arithmetic expression such as a + b are integers, the expression is called an **integer expression**, and the operation is called **Integer Arithmetic.**

Example:

a = 14 and b = 4 we have the following results:

a - b = 10

a + b = 18

a * b = 56

a / b = 3 (decimal part truncated)

a % b = 2 (remainder of integer division)

Real Arithmetic

- An arithmetic operation involving only real operands is called **Real Arithmetic**.
- A real operand may assume values either in decimal or exponential notation.

Example Program:

```
class FloatPoint
  {
    public static void main (String args[])
   {
        float a = 20.5F, b = 6.4F;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" a+b = " + (a+b) ) ;
        System.out.println(" a-b = " + (a-b) ) ;
        System.out.println(" a*b = " + (a*b) ) ;
        System.out.println(" a/b = " + (a/b» ;
        System.out.println(" a%b = " + (a%b) ) ;
   }
}
```

Output of the Program

a = 20.5

b = 6.4

a+b = 26.9

a-b = 14.1

a*b = 131.2

a/b = 3.20313

a%b = 1.3

Mixed-mode Arithmetic

- One of the operands is real and the other is integer, the expression is called a **mixed-mode Arithmetic** expression.

Example:

15/10.0 produces the result 1.5

Where as

15/10 produces the result 1

Relational Operators

- We may compare the age of two persons, or the price of two items, and so on. These Comparisons can be done with the help of relational operators.

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Example Program

```
class RelationalOperators
{
    public static void main (String args[])
   {
        float a = I5.0F, b = 20.75F, c = 15. OF;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b) ;
        System.out.println(" c = " + c);
        System.out.println(" a < b is " + (a <b));
        System.out.println(" a > b is " + (a>b));
        System.out.printlrr(" a = = c:is " + (a= =c));
```

```
        System.out.println(" a <= c is " + (a<=c));
        System.out.println(" a >= b is " + (a>=b));
        System.out.println(" b != c is " + (b!=c));
        System.out.println(" b = = a+c is " + (b= =a+c) ) ;
    }
}
```

Output of the Program

a = 15 a >= b is false

b - 20.75 b != c is true

c = 15 b = = a+c is false

a < b is true

a > b is false

a = = c is true

a <= c is true

Logical Operators

- Java has three logical operators.
- An expression is combines two or more relational expressions is termed as a **logical expression**

Operator	**Meaning**
&& | | !	logical AND logical OR logical NOT

Truth Table

op-l	**op-2**	**Value of the expression**	
		op-l && op-2	**op-l || op-2**
true	true	true	true
true	false	false	True
false	true	false	True
false	false	false	false

Assignment Operators

- **Assignment operators** are used to assign the value of an expression to a variable.
- We have seen the usual assignment operator, ‘ = ’.
- The general form is:

V op = exp;

Example:

A=5;

Tot=m1+m2+m3;

Increment and decrement operators

- These are the increment and decrement operators:

++ and --

- The operator **+ + adds. 1** to the operand while **- - subtracts 1**.

Both are unary operators and are used in the following form:

++m or m++

--m or m--

++m; is equivalent to m = m + 1; (or m += 1;)

--m; is equivalent to m = m - 1; (or m -= 1;

Example Program:

```
class IncrementOperator
{
    public static void main(String args[])
   {
        int m = 10, n = 20;
        System.out.println(" m = " + m);
        System.out.println(" n = " + n);
        System.out.println(" ++m = " + ++m);
        System.out.println(" n++ = " + n++);
        System.out.println(" m = " + m);
        System.out.println(" n = " + n);
   }
}
```

Output of the Program

m = 10

n = 20

++m = 11

n++ = 20

m = 11

n = 21

Conditional Operator

- The character pair ?: is a ternary operator available in Java. This operator is used to construct conditional expression of the form

exp1 ? exp2 : exp3

where exp1, exp2, and exp3 are expressions.

For example, consider the following statements:

a = 10;

b = 15;

x = if (a > b) ? a : b;

In this example, x will be assigned the value of b. This can be achieved using the if.•.else

statement as follows:

if(a > b)

x = a;

else

x = b;

Bitwise Operators

- Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level.

Operator	Meaning
&	bitwise AND
!	bitwise OR
^	bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right
>>>	shift right with zero fill

Special Operators

- Java supports some special operators of interest such as **instance of** operator and **member selection operator (.).**
- ***instanceof Operator***
- The instanceof is an object reference operator and returns true, if the object on the left-hand side is an instance of the class given on the right-hand side.
- **Example:**
    - person instanceof student
- ***Member Selection Operator or Dot Operator (.).***
- The dot operator (.) is used to access the instance variables and methods of class objects.
- **Examples:**
    - personl.age / / Reference to the variable age
    - personl.salary( ) / / Reference to the method salary( )

Arithmetic Expressions

- An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language.
- **Examples:**

Algebraic expression	Java expression
a b – c	a*b-c
(m+n)(x+y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2 + 2x + 1$	3*x*x+2*x+1
$\frac{x}{y} + c$	x/y+c

Evaluation of Expression

- Expressions are evaluated using an assignment statement of the form

  variable = expression;

- The expression is evaluated first and the result then replaces the previous value of the variable on the left-hand Side.

- **Examples:**
    - x = a*b-c;
    - y = b/c*a;
    - z = a-b/c+d;

Precedence of Arithmetic Expressions

- An arithmetic expression without any parentheses will be evaluated from **left to right** using the rules of precedence of operators.
- There are two distinct **priority levels** of arithmetic operators in Java:
    - High priority - *, / , %
    - Low priority - + , -

Example:

x = a-b/3+c*2-1

When **a = 9, b = 12, and c = 3**, the statement becomes

x = 9-12/3+3*2-1

and is evaluated as follows:

First pass

Step1: x = 9-4+3*2-1 (12/3 evaluated)

Step2: x = 9-4+6-1 (3*2 evaluated)

Second pass

Step3: x = 5+6-1 (9-4 evaluated)

Step4: x = 11-1 (5+6 evaluated)

Step5: x = 10 (11-1 evaluated)

Type Conversion in Expression

- One operator at a time involving two operands.
- If the operands are of different types, the 'lower' type is automatically converted to the higher' type before the operation proceeds.
- The result is of the higher type.

Table 5.9 Automatic Type Conversion Chart

	char	byte	short	int	long	float	double
char	int	int	int	int	long	float	double
byte	int	int	int	int	long	float	double
short	int	int	int	int	long	float	double
int	int	int	int	int	long	float	double
long	long	long	long	long	long	float	double
float	float	float	float	float	float	float	double
double	double	double	double	double	double	double	Double

Examples	Action
x = (int) 7.5	7.5 is converted to integer by truncation
a = (int)21.3/(int)4.5	Evaluated as 21/4 and the result would be 5
b = (double)sum/n	Division is done in floating point mode.
y = (int) (a+b)	The result of a + b is converted to integer.
z = (int) a+b	a is converted to integer and then added to b.
p = cost«double)x)	Converts x to double before using it as parameter.

Example Program:

```
class Casting
{
    public static void main (String args[])
   {
        flat sum;
        int i;
        sum = O. OF;
        for(i = 1; i <= 10; i++)
        {
            sum = sum + l/(float)i;
           System.out.println("      i = " +  i + sum = " + sum);
```

```
        }
    }
}
```

Output of the Program

i = 1 sum = 1

i = 2· sum = 1. 5

i.= 3 sum = 1. 83333

i = 4 sum = 2.08333

i = 5 ~um = 2.28333

i = 6 sum = 2.45

i = 7 sum = 2'.59286

i = 8 sum = 2.71786

i = 9 sum = 2.82897

i = 10 sum = 2.92897

Operator Precedence and Associatively

- Each operator in Java has a precedence associated with it.
- This precedence is used to an expression involving more than one operator is evaluated.
- The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from **left to right** or from **right to left**, depending on the level.
- This is known as the associativity

Table 5.11 Summary of Java Operaton

Operator	Description	Associativity	Rank
. () []	Member selection Function call Array element reference	Left to right	1
- ++. -- ! - (type)	Unary minus Increment Decrement Logical negation Ones complement Casting	Right to left	'2
* / %	Multiplication Division Modulus	Left to right	3
+ -	Addition Subtraction	Left to right	4
« » »>	Left shift Right shift Right shift with zero fill	Left to right	5
< <= > >= instanceof	Less than Less than or equal to Greater than Greater than or equal to Type comparison	Left to right	6
— !=	Equality Inequality	Left to right	7

Operator	Description	Associativity	Rank
&	Bitwise AND	Left to right	8
··	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional operator	Right to left	13
= op=	Assignment operators Shorthand assignment	Right to left	14

Mathematical Functions

- Mathematical functions such as cos, sqrt, log, etc. are frequently used in analysis of real life problems.
- Java supports these basic math functions through Math class defined in the **java.lang package.**

Example:

double y = Math.sqrt(x);

Functions	Action
sin (x)	Returns the sine of the angle x in radians
cos(x)	Returns the cosine of the angle x in radians
tan (x)	Returns the cosine of the angle x in radians
asin(y)	Returns the angle whose sine is y
acos(y)	Returns the angle whose cosine is y
atan(y)	Returns the angle whose tangent is y
atan2(x,y)	Returns the angle whose tangent is x/y
pow(x,y)	Returns x raised to y (xr)
exp(x)	Returns e raised to x (ex)
log (x)	Returns the natural logarithm of x
sqrt(x)	Returns the square root of x
ceil (x)	Returns the smallest whole number greater than or equal to x. (Rounding up)
iloor(x)	Returns the largest whole number less than or equal to x (Rounded down)
rint(x)	Returns the truncated value of x.
abs(a)	Returns the absolute value of a
max(a,b)	Returns the maximum of a and b
min(a,b)	Returns the minimum of a and b

ERODE ARTS AND SCIENCE COLLEGE (AUTONOMOUS)-ERODE

DEPARTMENT OF COMPUTER APPLICATIONS

PROGRAMMING IN JAVA

CLASS: II BCA STAFF NAME: M.DIVYA, M.SC, M.PHIL

UNIT – III

DECISION MAKING AND BRANCHING, LOOPING AND CLASSES

Decision Making with If Statement – Simple If Statement - If- Else Statement, Nesting of If-Else Statement - Else If Statement - Switch -? . Looping: While, Do-While, for, Comma Statements – Continue Classes: - Creating Object, Accessing Class Members - Constructors Methods of Overloading - Static Members - Nesting of Methods-Inheritance - Overriding Methods - Final Variables and Methods - Final Classes - Finalized Methods - Abstract Methods and Classes - Visibility Control. Arrays, Strings and Vectors: Creating an Array Two Dimensional Arrays - Strings - Vectors - Wrapper Classes.

DECISION MAKING AND BRANCHING, LOOPING AND CLASSES

➢ A program breaks the sequential flow and jumps to another part of the code, it is called **branching.** The branching is based on a particular condition, it is known as conditional branching.

➢ If branching takes place without any decision, it is known as unconditional branching.

➢ Java language possesses such decision making capabilities and supports the following statements known as **control or decision making** statements to implement branching.

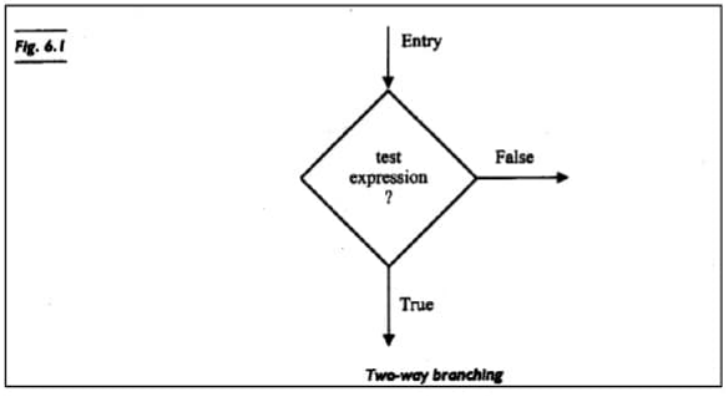
1. if statement
2. switch statement
3. Conditional operator statement

If statement

➢ The if statement is a powerful decision making statement and is used to control the flow of execution of statements.

```
if (test expression)
```

➢ It allows the computer to evaluate the expression first and value of the expression (relation or condition) is 'true' or 'false', it transfers the control to a particular statement.



The if statement may be implemented in different forms depending on the complexity of conditions to be tested. 1. Simple if statement

2. *if .. else statement*
3. *Nested if .. else statement*
4. *else if ladder*

6.3 SIMPLE IF STATEMENT

The general form of a simple if statement is

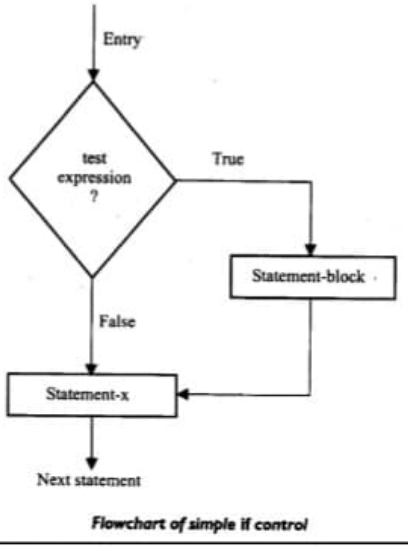
```
if(test expression)
{
    statement-block;
}
statement-x;
```

Simple if statement

➢ *The '**statement-block**' may be a single statement or a group of statements.*

➢ *If the test expression is **true**, the statement-block will be executed; otherwise the statement-block will be **skipped** and the execution will jump to the statement-x.*

Fig. 6.2



Flowchart of simple if control

Example Program:

```
public class IfStatementExample
{

public static void main(String args[])
{     int num=70;
if( num < 100 )
{
     /* This println statement will only execute,
      * if the above condition is true
      */
     System.out.println("number is less than 100");
}
System.out.println("number is More than 100");
}
}
```

Output:

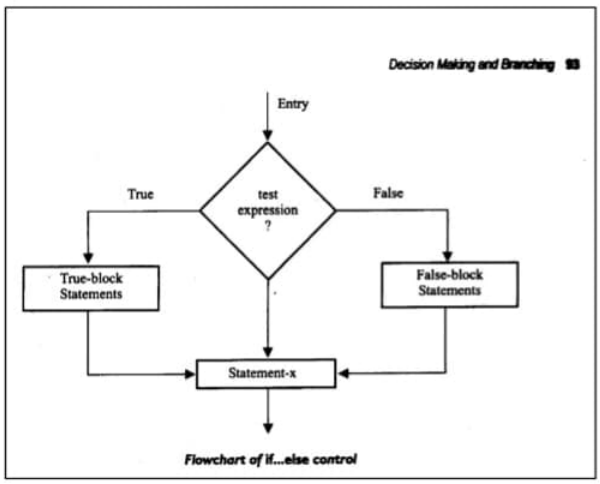
```
number is less than 100
```

If .. else statement

The if...else statement is an extension of the simple if statement. The **general form** is if (test expression)

```
{
True-block statement(s);
} else
{
False-block statement(s);
} statement-x;
```

➢ If the **test expression** is true, then the true-**block statement**(s) are executed; ➢ Otherwise, the **false-block statement**(s) are executed.



Flowchart of if...else control

Example Program

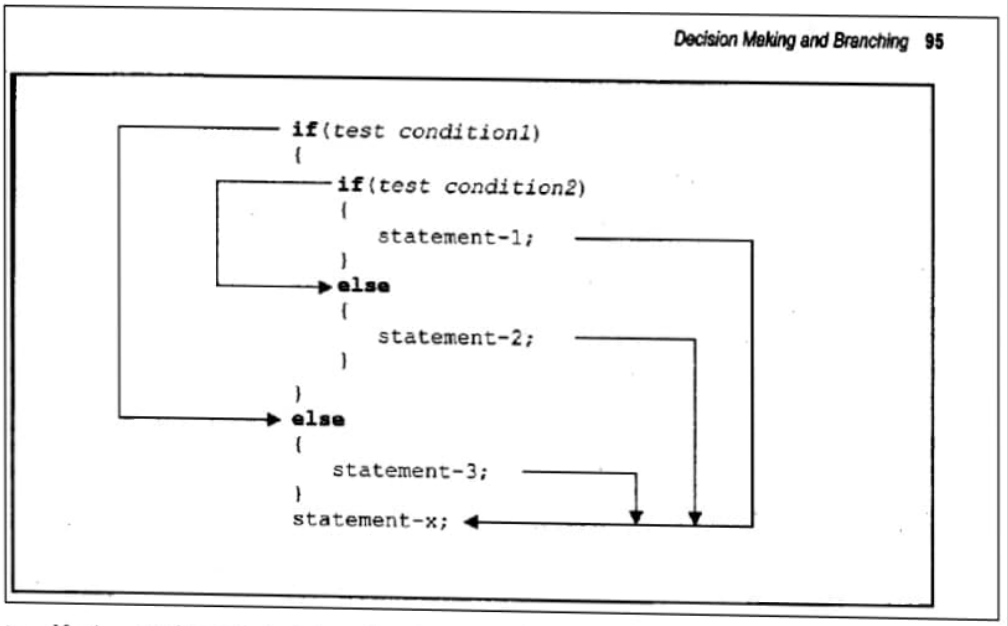
```
public class IfElseExample
{

  public static void main(String args[])
  {
    int testscore=70;
    if(testscore> 50)
    {
          System.out.println("The Result is PASS   ");
    }
    else
    {
          System.out.println("The Result is FAUL   ");
    }
  }
}
```

Output:

The Result is PASS

NESTING OF IF...ELSE STATEMENTS

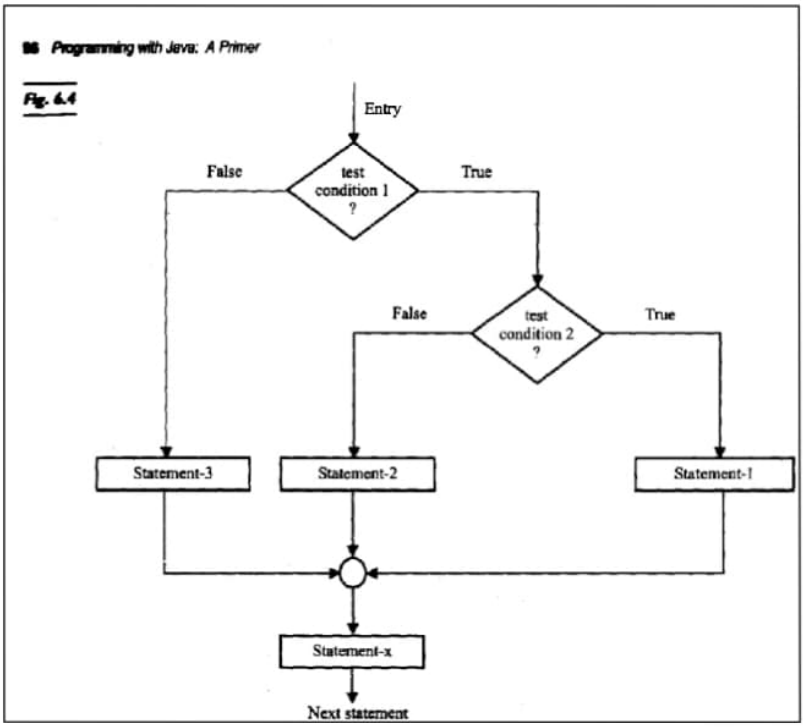
A series of decisions are involved, we may have to use more than one if...else statement in nested form as follows:



➢ If the **condition-1** is false, the statement-3 will be executed; otherwise it continues to perform the second test.

➢ If the **condition-2** is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x.

➢



Example Program

```
public class NestingIfExample
{

public static void main(String args[])
{    int a=70;
if(a > 0 )  // Condition - 1
{
if(a == 0 ) // Condition - 2
{

        System.out.println("A value is equal to ZERO");
}       else        {
        System.out.println("A value is Positive value");
}    else    {
     System.out.println("A value is Negative value");
}
}
}
```

Output:

A value is Positive value

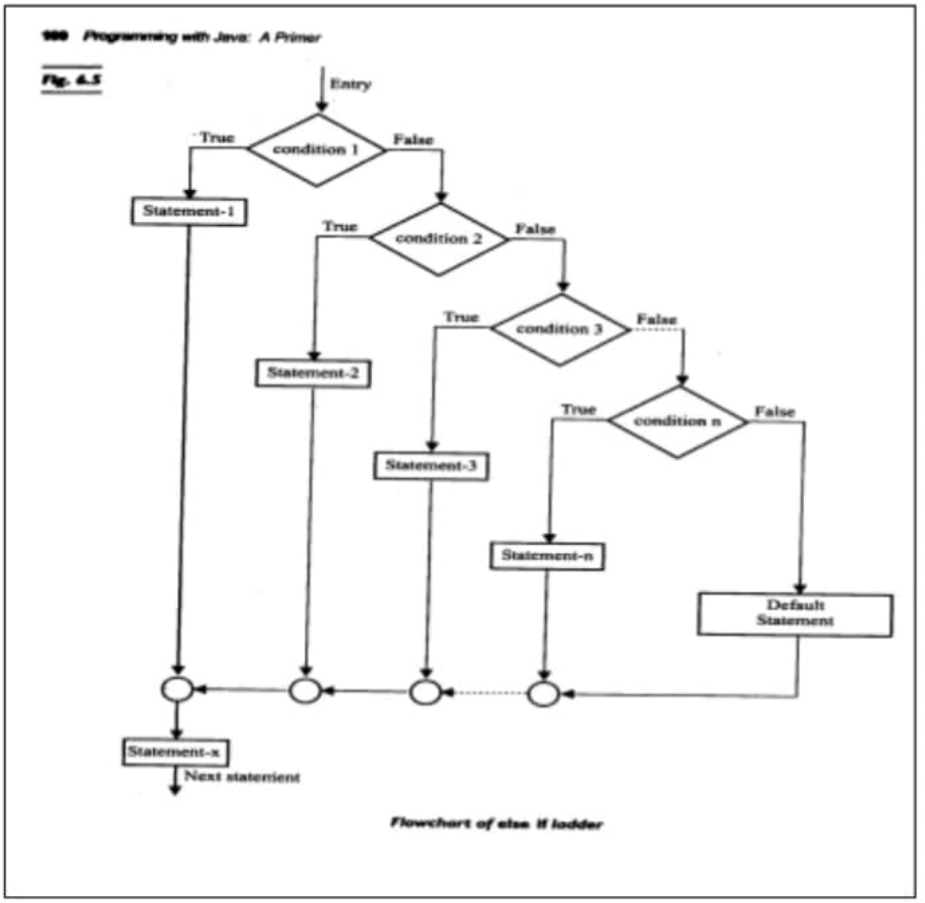
else if ladder

A multipath decision is a chain of ifs in which the statement associated with each else is an if. It takes the following ***general form***

```
if(condition1)
   statement-1;

else if(condition 2)
   statement-2;

     else if(condition 3)
        statement-3;
        ..................

          else if (condition n)
                  statement-n;

               else
                 default-statement;

statement-x;
```

➢ *This construct is known as the* ***else if ladder****. The conditions are evaluated from the top (of the ladder), downwards.*

➢ *As soon as the true condition is found, the statement associated with it is executed and the control is transferred to the statement*



Flowchart of else if ladder

Example Program

```
public class ControlFlowDemo
{
public static void main(String[] args)
{       char ch = 'o';

if (ch == 'a' // ch == 'A')          System.out.println(ch + " is vowel.");       else if (ch == 'e' // ch == 'E')
System.out.println(ch + " is vowel.");       else if (ch == 'i' // ch == 'I')          System.out.println(ch + " is
vowel.");       else if (ch == 'o' // ch == 'O')          System.out.println(ch + " is vowel.");       else if (ch == 'u' //
ch == 'U')          System.out.println(ch + " is vowel.");       else
System.out.println(ch + " is Not a Vowel.");
    }
}
```

OUTPUT ======

o is vowel.

THE SWITCH STATEMENT

The switch statement is a multi-way branch statement.

It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

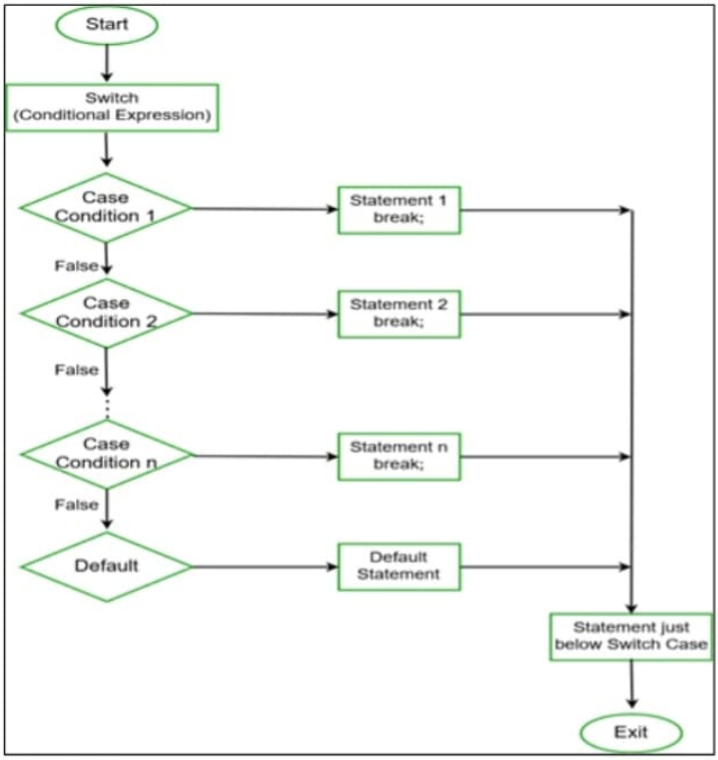
We have seen that when one of the many alternatives is to be selected, we can design a program using if statements to control the selection.

Use the switch statement to select one of many code blocks to be executed.

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- The break and default keywords are optional.

Syntax:

```
switch (expression){          case value-1:          block-1          break;
case value-2:          block-2          break;
        .....
        .....          default:          default-block
                break;
    }
    statement-x;
```




```
public class Test {
public static void main(String[] args)
{      int day = 5;      String dayString;

// switch statement with int data type      switch (day) {      case 1:
dayString = "Monday";          break;      case 2:
dayString = "Tuesday";         break;      case 3:
dayString = "Wednesday";       break;      case 4:
dayString = "Thursday";        break;      case 5:      dayString = "Friday";      break;      case 6:
dayString = "Saturday";        break;      case 7:
dayString = "Sunday";          break;      default:      dayString = "Invalid day";      break;
}
System.out.println(dayString);
}
}
```

Output:

Friday

DECISION MAKING AND LOOPING

Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

1. The while statement
2. The do statement 3. The for statement

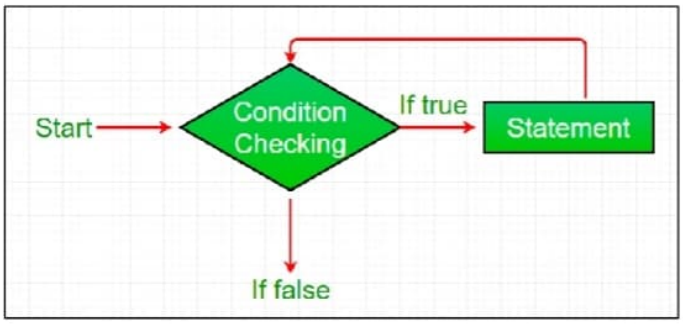
1. WHILE LOOP:

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

The while loop can be thought of as a repeating if statement.

Syntax:

```
while (boolean condition)
{   loop statements...
}
```



While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**

➢ Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.

➢ When the condition becomes false, the loop terminates which marks the end of its life cycle.

➢ Example program

➢ Example program class whileLoopDemo

```
{
public static void main(String args[])
{       int x = 1;

// Exit when x becomes greater than 4        while (x <= 4)
{
System.out.println("Value of x:" + x);

// Increment the value of x for          // next iteration          x++;       }
}
}
```

Output:

Value of x:1

Value of x:2

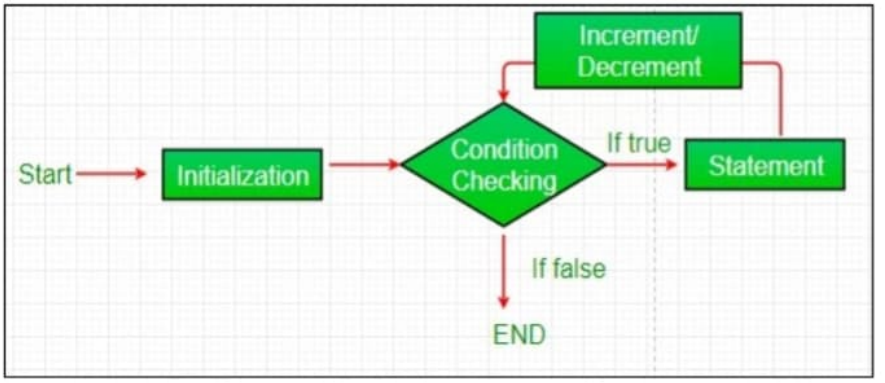
Value of x:3

2. FOR LOOP:

For loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping. **Syntax:**

```
for (initialization condition; testing condition;                    increment/decrement)
{    statement(s) }
```

Flowchart:



➢ ***Initialization condition:*** *Here, we initialize the variable in use. It marks the start of a for loop.*

An already declared variable can be used or a variable can be declared, local to loop only.

➢ ***Testing Condition:*** *It is used for testing the exit condition for a loop. It must return a boolean value. It is also an* ***Entry Control Loop*** *as the condition is checked prior to the execution of the loop statements.*

➢ ***Statement execution:*** *Once the condition is evaluated to true, the statements in the loop body are executed.*

➢ ***Increment/ Decrement:*** *It is used for updating the variable for next iteration.*

➢ ***Loop termination:****When the condition becomes false, the loop terminates marking the end of its life cycle.*

Example program class forLoopDemo

```
{
public static void main(String args[])
{
// for loop begins when x=2          // and runs till x <=4          for (int x = 2; x <= 4; x++)
System.out.println("Value of x:" + x);
}
}
```

Output:

Value of x:2

Value of x:3

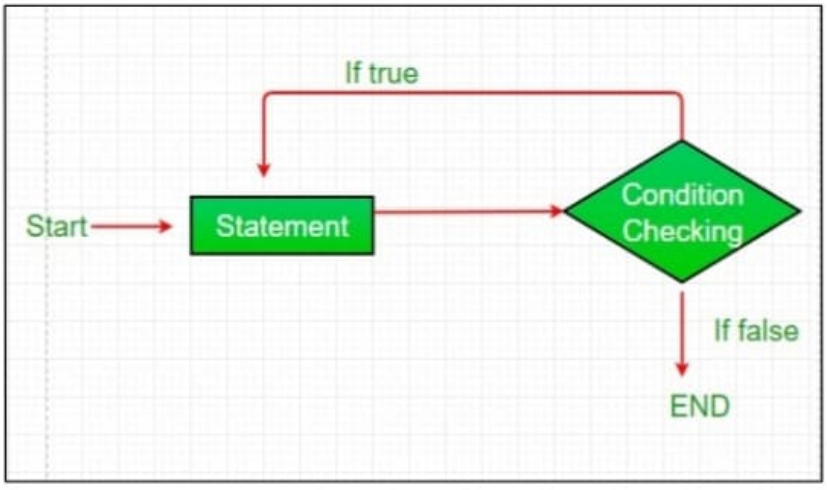
Value of x:4

3.DO WHILE:

Do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of ***Exit Control Loop.***

Syntax:

```
do {    statements..
}
while (condition);
```



➢ *do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.*

➢ *After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.*

➢ *When the condition becomes false, the loop terminates which marks the end of its life cycle.*

➢ *It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.*

➢ *Example program*

```
class dowhileloopDemo
{
public static void main(String args[])
{       int x = 21;         do
{
// The line will be printed even
// if the condition is false
System.out.println("Value of x:" + x);
x++;
}       while (x < 20);
}
}
```

Output:

Value of x: 21

JUMPS IN LOOPS

THE BREAK STATEMENT

Under some circumstances, we require the loop be exited even when the loop's testexpression still evaluates to true. Break statement used in these situations.

Syntax:

break;

example:

```
while(test-expression1)
{        s1;        s2;
    if(test-expression2)
break;
```

```
s3; s4;
}
```

THE CONTINUE STATEMENT

The continue statement causes skipping of the statements following it in the body of the loop, and also causes the control to be transferred back to the beginning of the loop.

Syntax: **Continue;**

Example:

```
while(test-expression1)
```

```
{   s1;        s2;
        if(test-expression2)
    ———     continue;
        s3;        s4;
    }
```

Difference between break and continue statements

The break statement	**the continue statement**
1. It can be used in switch statement.	1. It cannot be used in switch statement
2. It causes premature exit of the loop Enclosing it.	2. It causes skipping of the statements following it in the body of the loop.
3. The control is transferred to the statement Following the loop.	3. The control is transferred back to the loop.
4. The loop may not complete the intended No.of iterations.	4. The loop completes the intended no.of iterations.

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition.

The number of times a loop is repeated is decided in advance and the test condition is written to achieve this.

LABLE

According to nested loop, if we put break statement in inner loop, compiler will jump out from inner loop and continue the outer loop again.

What if we need to jump out from the outer loop using break statement given inside inner loop? The answer is, we should define **lable** along with colon(:) sign before loop

CLASSES:

- Java is an object-oriented programming language.
- Everything in Java is associated with classes and objects, along with its attributes and methods.
- For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- A Class is like an object constructor, or a "blueprint" for creating objects.

CREATE A CLASS :

To create a class, use the keyword class:

General format

class classname [extends superclass name]

```
{
  [variable declaration;]
  [methods declaration;]
}
```

Adding variables:

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are calls as ***instance variables***.

Ex: Class Rectangle

```
{
    int length;  int width;
        }
```

Adding methods:

Method declarations have four basic parts:

- The name of the method (method name)
- The type of the value the method returns (type)
- A list of parameters (parameter-list)
- The body of the method

Ex: class rectangle

```
{
     int length;       int width
         void getdata (int x, int y)
         {
           Length = x;
           Width = y;
            }
        }
```

CREATING OBJECTS

Objects in java are created using **new** operator, which creates an object of the specified class and returns a reference to the variable.

Ex : Rectangle rect1 = *new* Rectangle ( );

Now that we have created objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program.

Remember, all variables must be assigned values before they are used.

Since we are outside the class, we cannot access the instance variables and the methods directly.

To do this, we must use the concerned object and the dot operator shown below

```
objectname. variable name
objectname . methodname (parameter-list) ;
```

Example

In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

Example program

Create an object called "*myObj*" and print the value of x:

```
public class MyClass {

int x = 5;  public static void main(String[] args) {

MyClass myObj = new MyClass();

System.out.println(myObj.x);

}

}
```

Output 5

ACCESSING CLASS MEMBERS:

· All variables must be assigned unique values before they are used. Since, we are outside the class, we cannot access the instance variables and methods directly. · To do this, we must use the concerned object and the dot operator as shown below: **Objectname.variablename;**

Objectname.methodname(parameter list);

Example:

```
class Square
{
    int side;
    void getdata(int s)
{ side=s; }
    int rectarea()      { int area=side* side;
        return(area);
    }
    }
    class Squarearea
    {
    public static void main(String arg[])
    {
        Square s1=new Square();
        Square s2=new Square();
        s1.side=5;
    int area1,area2;        area1=s1.side*s1.side;        s2.getdata(5);
area2=s2.rectarea();
        System.out.println(area1);
        System.out.println(area2);
    }
}
```

CONSTRUCTORS

Java supports a special type of method called a CONSTRUCTOR that enables an object to initialize itself when it is created.

- Constructor has the same name as the class itself.
- They do not specify a return type not even void.
- The constructors initialize an object immediately. The constructors have some conditions.
  - ✓ The constructors share the same name of the class
  - ✓ The constructor does't return any thing (void)
  - ✓ The constructors are automatically called during the creation of the objects

Example 1

```
class rectangle {
Int length ;
Int width ;
Rectangle (int x, int y) //constructor method
{
Length=x;
Width=y;
}
Int rectArea ( )
{
Return (length * width);
}
}
```

Example 2

```
Class volume
{ int x,y,z; volume() { x=10; y=10;
z=10;
} public int getdata()
{ int vol=x*y*z;
return vol;
}
} class demo {
public static void main(String args[])
{
volume obj=new volume() // Object declaration
int vol1=obj.getdata();
System.out.println("The volume is :"+vol1);
}
}
```

Parameterized Constructors

- If the initializations of the variables are not very useful all the volume have the same value until the completion of the program.
- The solution is to add parameter to the constructor for the various dimensions.
- This makes more effective programmer then the initialization of the values through the constructors. Example

```
class volume
{
int x,y,z;
volume(int m,int n,int O)
{          x=m;          y=n;          z=O;
}          public int getdata()
{          int volu=x*y*z;          return volu;
}
}
class demo
{
public static void main(String args[])
{
volume obj=new volume(100,200,300);//Parametirized Constructors          int vol=obj.getdata();
System.out.println("The volume is :"+vol);
}
}
```

USING THE THIS KEYWORD

Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called.

You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

The most common reason for using the keyword is because a field is shadowed by a method or constructor parameter.

- **This** can be used inside any method to refer to the current object.
- **This** is always a reference to the object on which the method was invoked.
- We can use **this** anywhere a reference to an object of the current class type is permitted. Ex: box(double w, double h, double d)

```
{
    this.width=w;      this.height=h;      this.depth=d;
}
```

Garbage collection

- Objects are dynamically allocated by using the new operator.
- Java handles deallocation automatically. The technique that accomplishes this is called garbage collection.

The finalize() method

- Sometimes an object will need to perform some action when it is destroyed.
- If an object is holding some non-java resource such as a file handle or window character font, then we might want to make sure these resources are freed before an object is destroyed.
- Java provides a mechanism called finalization. By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- The finalize method has this general form: protected void finalize()

```
{
    //finalization code
}
```

Here, the keyword protected is a specifier that prevents access to finalize() by code defined outside its class.

OVERLOADING IN JAVA

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.

- In java, it is possible to create methods that heave the same name but different parameter lists and different definitions. This is called method overloading.
- It is used when objects are required to perform similar task but using different input parameters.

When we call a method in an object, java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute

Example program class Room

{ int length; int breadth;' void getdata(int l,int b)

{ length=l; breadth=b;

```
System.out.println(length);
System.out.println(breadth);
} void getdata(int l)
{ length=breadth=l;
System.out.println(length);
System.out.println(breadth);
} void getdata() {
length=breadth=l;
System.out.println(length);
        System.out.println(breadth); } void getdata()
{ length=breadth=0;
System.out.println(length);
System.out.println(breadth);
} } class H { public static void main(String args[])
{
Room o1=new Room();
o1.getdata(2,3); o1.getdata(2); o1.getdata();
}
}
```

STATIC MEMBERS:

· A class basically contains two sections. One declare variables and the other declares methods. These variables and methods are called Instance variables and methods.

· This is because every time the class is instantiated a new copy of them is created. They are accessed using the objects.

· Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object.

· That is the member belongs to the class as a whole, rather than the objects created from the class.

· Such members can be defined as follows:

```
static int count; static int max(int x)
```

· The members that are declared as static are called static members. Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as **class variables and class methods.**

(i). Static variables can be called without using the objects. They can be called using class names.

(ii). They are also available for use by other classes

NESTING OF METHODS

A method can be called by using only by an object of that class. There is an exception to this. A method can be called by using only its name by another method of the same class. This is known as nesting of methods.

INHERITANCE

Inheritance is an important pillar of OOP(Object Oriented Programming).

It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Important terminology:

- ***Super Class***: The class whose features are inherited is known as super class(or a base class or a parent class).
- ***Sub Class***: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- ***Reusability***: Inheritance supports the concept of "reusability", i.e. 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 our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

The keyword used for inheritance is **extends**.

Syntax :

```
class derived-class extends base-class
{
//methods and fields
}
```

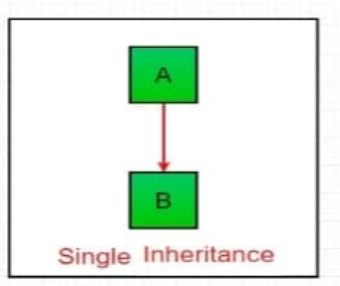
Types of Inheritance in Java Below are the different types of inheritance which is supported by Java.

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many subclasses)
- Multilevel inheritance (Derived from a derived class)

SINGLE INHERITANCE :

In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

```
import java.util.*;  import java.lang.*;
import java.io.*;

class one {     public void print_geek()
{
System.out.println("Geeks");
}
}

class two extends one
```




```
{     public void print_for()
{
System.out.println("for");
}
} // Driver class  public class Main
{
public static void main(String[] args)
{
two g = new two();         g.print_geek();
g.print_for();
g.print_geek();
}
}
```

Output:

Geeks

for

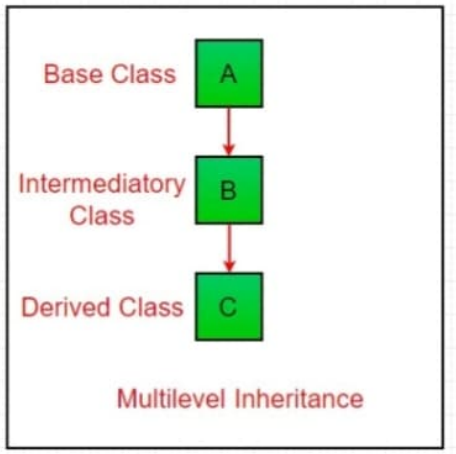
Geeks

MULTILEVEL INHERITANCE :

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

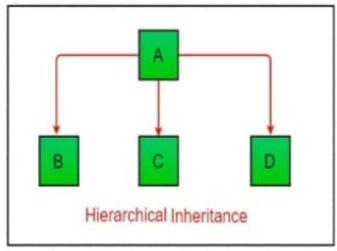
In Java, a class cannot directly access the grandparent's members.




```
import java.util.*; import java.lang.*;
import java.io.*;

class one {     public void print_geek()
{
System.out.println("Geeks");
}
}

class two extends one
{     public void print_for()
{
System.out.println("for");
}
}

class three extends two {     public void print_geek()
{
System.out.println("Geeks");
}
}

// Drived class  public class Main
{     public static void main(String[] args)
{         three g = new three();         g.print_geek();
g.print_for();
g.print_geek();
}
}
```

HIERARCHICAL INHERITANCE :

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.



Output:

Geeks
for
Geeks

```
import java.util.*; import java.lang.*; import java.io.*;

class one {
public void print_geek()
{
System.out.println("Geeks");
}
}

class two extends one
{
public void print_for()
{
System.out.println("for");
}
}

import java.util.*; import java.lang.*; import java.io.*;

class one {    public void print_geek()
{
System.out.println("Geeks");
}
}

class two extends one
{    public void print_for()
{
System.out.println("for");
}
}

class three extends one
{
/*.....*/
}
```

```
// Drived class  public class Main
{    public static void main(String[] args)
{       three g = new three();       g.print_geek();       two t = new two();       t.print_for();
g.print_geek();
}
}
```

Output:

Geeks for

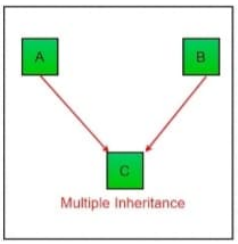
Geeks

MULTIPLE INHERITANCE (THROUGH INTERFACES) :

In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes.

Please note that Java does **not** support multiple inheritance with classes.

In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.




```
import java.util.*;  import java.lang.*;
import java.io.*;

interface one  {
public void print_geek();
}

interface two  {
public void print_for();
}

interface three extends one,two
{
public void print_geek();
}
class child implements three
{
@Override    public void print_geek() {       System.out.println("Geeks");
}
public void print_for()
{
System.out.println("for");
}
}
```

```
// Drived class  public class Main
{
public static void main(String[] args)
{       child c = new child();        c.print_geek();
c.print_for();
c.print_geek();
} }
```

Output:

Geeks

for

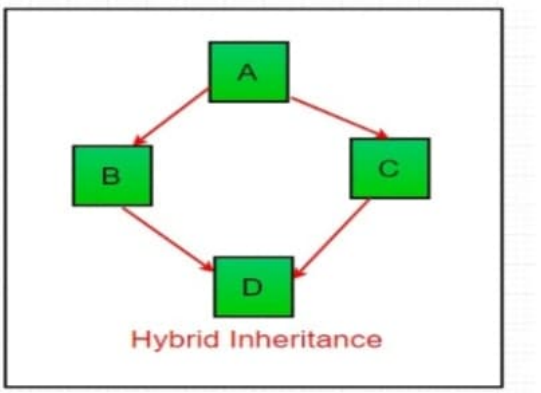
Geeks

HYBRID INHERITANCE (THROUGH INTERFACES) :

It is a mix of two or more of the above types of inheritance.

Since java doesn't support multiple inheritances with classes, the hybrid inheritance is also not possible with classes.

In java, we can achieve hybrid inheritance only through Interfaces.



METHOD OVERRIDING IN JAVA

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding.

Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.

In this case the method in parent class is called overridden method and the method in child class is called overriding method.

In this guide, we will see what is method overriding in Java and why we use it

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human.

The `Boy` class extends `Human` class. Both the classes have a common method `void eat()`.

Boy class is giving its own implementation to the `eat()` method or in other words it is overriding the `eat()` method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
lass Human{
  //Overridden method
  public void eat()
  {
    System.out.println("Human is eating");
  }
}
class Boy extends Human{
  //Overriding method
  public void eat(){
    System.out.println("Boy is eating");
  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
  }
}
```

Output:

VISIBILITY CONTROL –

Sometimes it is necessary to restrict the access to certain variables and methods from outside the class.

We can achieve this in Java by applying visibility modifiers to the instance variables and methods. The visibility modifiers are also known as **access modifiers.**

Java provides three types of visibility modifiers: **public, private and protected.**

Public Access - A variable or method declared as public has the widest possible visibility and accessible everywhere.

Example:

public int number;

public void sum( ) { ..... }

Friendly Access –

When no access modifier is specified, the member defaults to a limited version of public accessibility known as friendly level of access.

The difference between the public access and the friendly access is that the public modifier makes fields visible only in the same package, but not in other packages

Protected access –

The visibility level of a protected field lies in between the public access and friendly access.

That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages.

Private access –

Private fields are accessible only within their own class.

They cannot be inherited by subclass and therefore not accessible in subclasses. It prevents the method from being sub classed.

Private protected access –

A field can be declared with two keywords private and protected together like:

Private protected access –

A field can be declared with two keywords private and protected together like:

private protected int cldeNumber;

This gives a visibility level between the protected access and private access.

This modifier makes the fields are not accessible by other classes in the same package.

Public Access

Any variable or method is visible to the entire class in which it is defined. What if we want to make it visible to all the classes outside this class? This is possible by simply declaring the variable or method as public.

Table 8.1 Visibility of Fields in a Class

Access location \ Access modifier →	public	protected	friendly (default)	private protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

ARRAYS:

- An array is a group of contiguous or related data items that share a common name.
- A particular value is indicated by writing a number called index number or subscript in brackets after the array name.
- Array is a group of continuous or related data items that share a common name.
- Syntax:
- array_name[value]; - Example:
- salary[10];

- ONE-DIMENSIONAL ARRAY
- CREATING AN ARRAY
- DECLARATION OF ARRAY
- CREATION OF ARRAY
- INITIALIZATION OF ARRAY
- ARRAY LENGTH
- TWO-DIMENSIONAL ARRAY
- VARIABLE-SIZE ARRAY

ONE DIMENSIONAL ARRAY

A list of items can be given one variable name using only one subscript and such a variable is called a single- subscripted or One- dimensional array

- **Express as: x[1],x[2].....x[n]**
- **The subscript of an array can be integer constants, integer variables like i, or expressions that yield integers.**

CREATING AN ARRAY

- Arrays must be declared and created in the computer memory before they are used.
- Creation of an array involves three steps:
  1. Declare the array
  2. Create memory locations
  3. Put values into the memory locations.

DECLARATION OF ARRAYS

Arrays in Java may be declared in two forms:

Form1

Form2

Example:

int number[ ]; float average[ ]; int[ ] counter; float[ ] marks;

CREATION OF ARRAYS

- After declaring an array, we need to create it in the memory.
- Java allows creating arrays using new operator only, as shown below.

Arrayname = new type[size];

Examples:

Number = new int[5];

Average = new float [10];

INITIALIZATION OF ARRAYS

- The final step is to put values into the array created. This process is known as initialization.
- This is done using the array subscripts as shown below.

Arrayname [subscript] = value;

Example:

number[0]=35; number[1]=40;

. . . . . .

number[n]=19;

- We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below.

Type arrayname[ ] = {list of values};

- The array initializer is a list of values separated by commas and surrounded by curly braces.

ARRAY LENGTH

- All arrays store the allocated size in a variable named length.
- To access the length of the array a using a.length.
- Each dimension of the array is indexed from zero to its maximum size minus one.
- Example:

```
int aSize = a.length;
```

Example:

```
Class NumberSorting
{
    Public static void main(String args[])
    {
    int num[] = {55, 40, 80, 65, 71};          int n = num.length;
    System.out.print("Given list:");
    For(int I = 0; I < n; i++)
    {
    system.out.print(" " + num[i]);
        }
    system.out.println("\n");          for(int I = 0; I < n; i++)
        {
            for (int j = I + 1; j < n; j++){
                if(num[i] < num[j])
                {
                int temp = num[i];                    num[i] = num[j];                    num[j] = temp;
                }
            }
        }
    system.out.print("Sorted list :");          For(int I = 0; I < n; i++){          system.out.print(" " + num[i]);
        }
        system.out.println(" ");
      }
    }
```

Output:

Given list : 55 40 80 65 71

Sorted list : 80 71 65 55 40

TWO DIMENSIONAL ARRAYS

- In mathematics, we represent a particular value in a matrix by using two subscripts such as v_{ij} . Here v denotes the entire matrix and v_{ij} refers to the value in the ith row and jth coloumn.
- Ex: v[4][3];
- For creating two-dimensional arrays, we must follow the same steps as that of simple arrays. We may create a two-dimensional array like this: Int myArray[][]; myArray = new int[3][4];

-This creates a table that can store 12 integer values, four across and three down.

VARIABLE SIZE ARRAYS

- Java treats multidimensional arrays as "arrays of arrays".
- It is possible to declare a two-dimensional array as follows: Int x[][] = new int[3][];

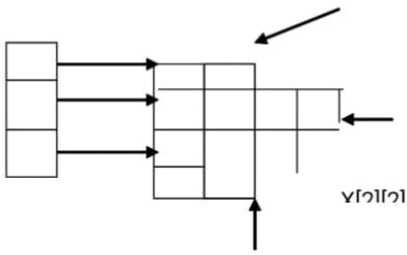
  X[0] = new int[2];

  X[1] = new int[4];

  X[2] = new int[3];
- These statements create a two-dimensional array as having different lengths for each row as shown below:

X[0][1]

X[0]



X[1] x[1][3]

X[2]

STRINGS ARRAYS

WRAPPER CLASSES IN JAVA

- Since, vectors cannot 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 packages.
- The wrapper classes have a number of unique methods for handling primitive data types and objects.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

UNIT 4

Interfaces, Packages of Multithreaded Programming

Interfaces : Multiple Inheritances

Introduction:

Java does not support multiple inheritance. That is, classes in Java cannot have more than one super class. For instance, a definition like

class A extends B extends C

{ .....

.....

}

is not permitted in Java. .

- A large number of real-life applications require the use of multiple inheritance. Since C+ + like implementation of multiple inheritance proves difficult and adds complexity to the language.
- Java provides an alternate approach known as interfaces to support the concept of multiple inheritances. Although a Java class cannot be a subclass of more than one super class, it can implement more than one interface,

Defining Interfaces

- An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference.
- The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants.
- The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is:

```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

Here, **interface** is the key word and *InterfaceName* is any valid Java variable (just like class names). Variables are declared as follows:

```
static final type VariableName = Value;
```

Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example:

```
return-type methodName1 (parameter_list);
```

Here is an example of an interface definition that contains two variables and one method

```
interface Item
{
    static final int code = 1001;
    static final String name = "Fan";
    void display ( ) ;
}
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon. The class that implements this interface must define the code for the method.

Another example of an interface is:

```
interface Area
{
    final static float pi = 3.142F;
    float compute (float x, float y);
    void show ( );
}
```

10.3 EXTENDING INTERFACES

Like classes, interfaces can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword **extends** as shown below:

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

For example, we can put all the constants in one interface and the methods in the other. This will enable us to use the constants in classes where the methods are not required. Example:

```
interface  ItemConstants
{
     int code = 1001;
     string name = "Fan";
}
interface Item extends ItemConstants
{
     void display ( );
}
```

10.4 IMPLEMENTING INTERFACES

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname implements interfacename
{
    body of classname
}
```

Here the class *classname* "implements" the interface *interfacename*. A more general form of implementation may look like this:

```
class classname  extends superclass
      implements interface1,interface2, ......
{
      body of classname
}
```

// InterfaceTestJava

interfaceArea // Interface defined

{

```
final static float pi = 3.14F;
float compute (float X, float y);
}
class Rectangle implements Area                // Interface implemented
{
        public float compute (float X, float y)
        {
                return (x*y) ;
        }
}
class Circle implements Area                    // Another implementation
{
        public float compute (float X, float y)
        {
                return (pi *x*x) ;
        }
}
class InterfaceTest
{
        public static void main (String args[ ])
        {
        Rectangle rect = new Rectangle( );
        Circle cir = new Circle( );
        Area area;              // Interface object
        area = rect;
        System.out.println("Area of Rectangle = " + area.compute(10,20));
        area = cir;
        System.out.println("Area of Circle = “ + area.compute(lO,O));
        }
```

}

Accessing Interfaces Variables:

- Interfaces can be used to declare a set of constants that can be used in different classes. This is similar to creating header files in C+ + to contain a large number of constants.
- The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value. Example:

```
class Student
{
        int rollNumber;
        void getNumber(int n)
        {
                rollNumber = n;
        }
        void putNumber( )
        {
        System.out.println(“Roll No : " + rollNumber);
        }
}
class Test extends Student
{
        float partl, part2;
        void getMarks(float ml, float .m2)
        {
                partl = ml;
                part2 = in2;
        }
        void putMarks( )
        {
```

```
                System.out.println("Marks obtained");
                System.out.println("Partl =" + partl);
                System.out.println("Part2 =" + part2);
        }
}
interface Sports
{
        float sportWt = 6.0F;
        void putWt( );
}
class Results extends Test implement Sports
{
        float total;
        public void putWt( )
        {
                System.out.println(~Sports Wt = " + sportWt);
        }
        void display ( )
        {
                total = part1 + part2 + sportWt;
                putNumber( );
                putMarks( );
                putWt ( );
                System.out.println(~Total score = " + total);
        }
}
class Hybrid
{
        public static void main (String args[ ])
```

```
    {
        Results student1 = new Results ( );
        student1.getNumber(1234);
        student1.getMarks(27.5F, 33.0F);
        student1.display( );
    }
}
```

Packages: Using System Packages

- Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes.
- By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.

2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.

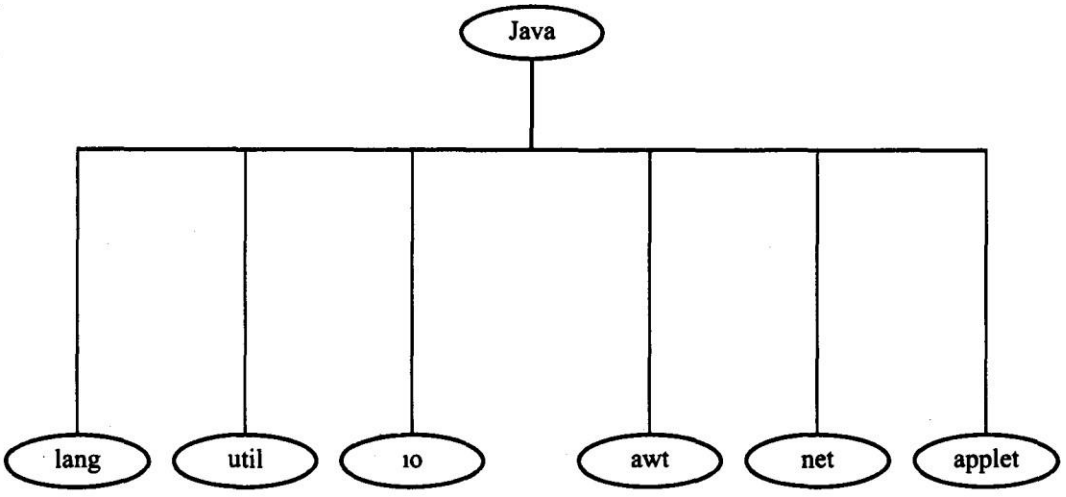
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.

4. Packages also provide a way for separating "design" from "coding".

- First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.
- Java packages are therefore classified into two types. The first category is known

  Java API packages and the second is known as **User defined packages**.

Fig. 11.1

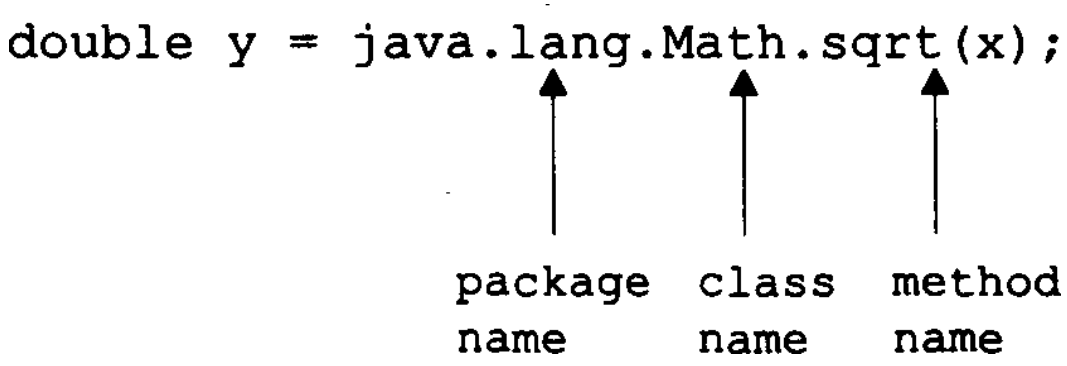


Java system packages

Package name	Contents
java.lang	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, date, 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.

Naming Conversions

- Packages can be named using the standard Java naming rules. By convention, packages begin with lowercase letters. This makes it easy for users to distinguish package names from class names when looking at an explicit reference to a class.
- All class names, again by convention, begin with an uppercase letter. For example, look at the following statement:



This statement uses a fully qualified class name **Math** to invoke the method **sqrt ().** Note that methods begin with lowercase letters. Consider another example:

java.awt.Point pts[ ];

Creating Packages

- Java system packages are organised and used. Now, let us see how to create our own packages.
- We must 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

Then we define a class, just as we normally define a class. Here is an example:

package firstPackage; // package declaration

public class FirstClass // class definition

{ .....

..... (body of class)

....

}

Creating our own package involves the following steps:

1. Declare the package at the beginning of a file using the form

2. Define the class that is to be put in the package and declare it public.

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

4. Store the listing as the classname.java file in the subdirectory created.

5. Compile the file. This creates .class file in the subdirectory.

Accessing Packages

- Java system package can be accessed either using a fully qualified class name or using a shortcut approach through the import statement.
- We use the import statement when there are many references to a particular package or the package name is too long.
- The same approaches can be used to access the user-defined packages as well.
- The import statement can be used to search a list of packages for a particular class. The general form of import statement for searching a class is as follows:

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

Adding A Class to A Packages

It is simple to add a class to an existing package. Consider the following package:

packcage pI;

public ClassA

{

/ / body of A

}

The package pI contains one public class by name A. Suppose we want to add another class B to this package. This can be done as follows:

1. Define the class and make it public.

2. Place the package statement

package pI;

before the class definition as follows:

package pI;

public class B

{

/ / bodyof B

}

3. Store this as B.java file under the directory pl.

4. Compile B.java file. This will create a B.cl88s file and place it in the directory pl.

If we want to create a package with multiple public classes hi it, we may follow the following steps:

1. Decide the name of the package.

2. Create a subdirectory with this name under the directory where main source files are stored.

3. Create classes that are to be placed in the package in separate source files and declare the package statement

package packagename;

at the top of each source file.

4. Switch to· the subdirectory created earlier and compile each source file. When completed, the package would contain .class files of all the source files.

Hiding Classes

- We import a package using asterisk (*), all public classes are imported.
- However, we may prefer to "not import" certain classes. That is, we may like to hide these classes from accessing from outside of the package.
- Such classes should be declared "not public". Example:

package pl;

public class X **/ / public class, available outside**

{

/ / bodyof X

}

class Y **/ / not public, hidden**

{

/ / body of Y

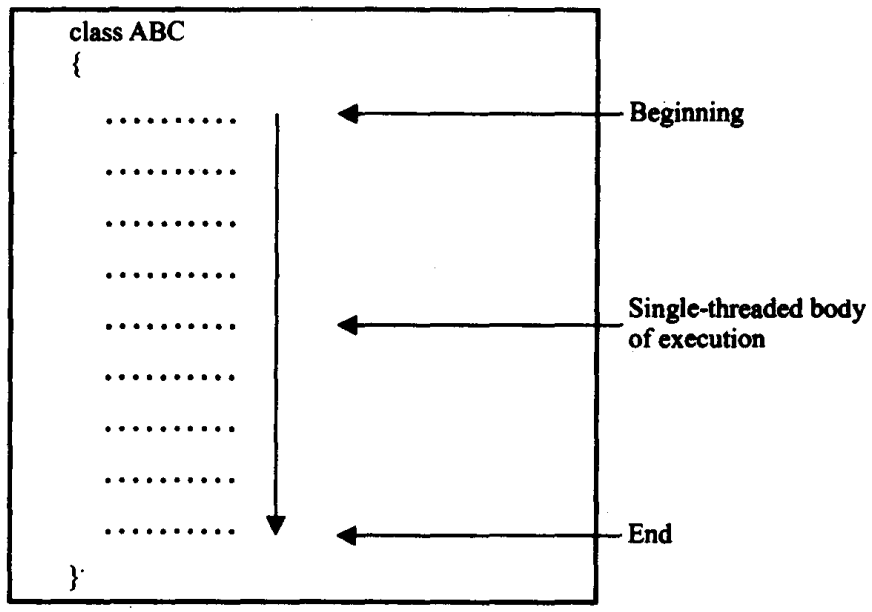
}

Multithreaded Programming:

Introduction:

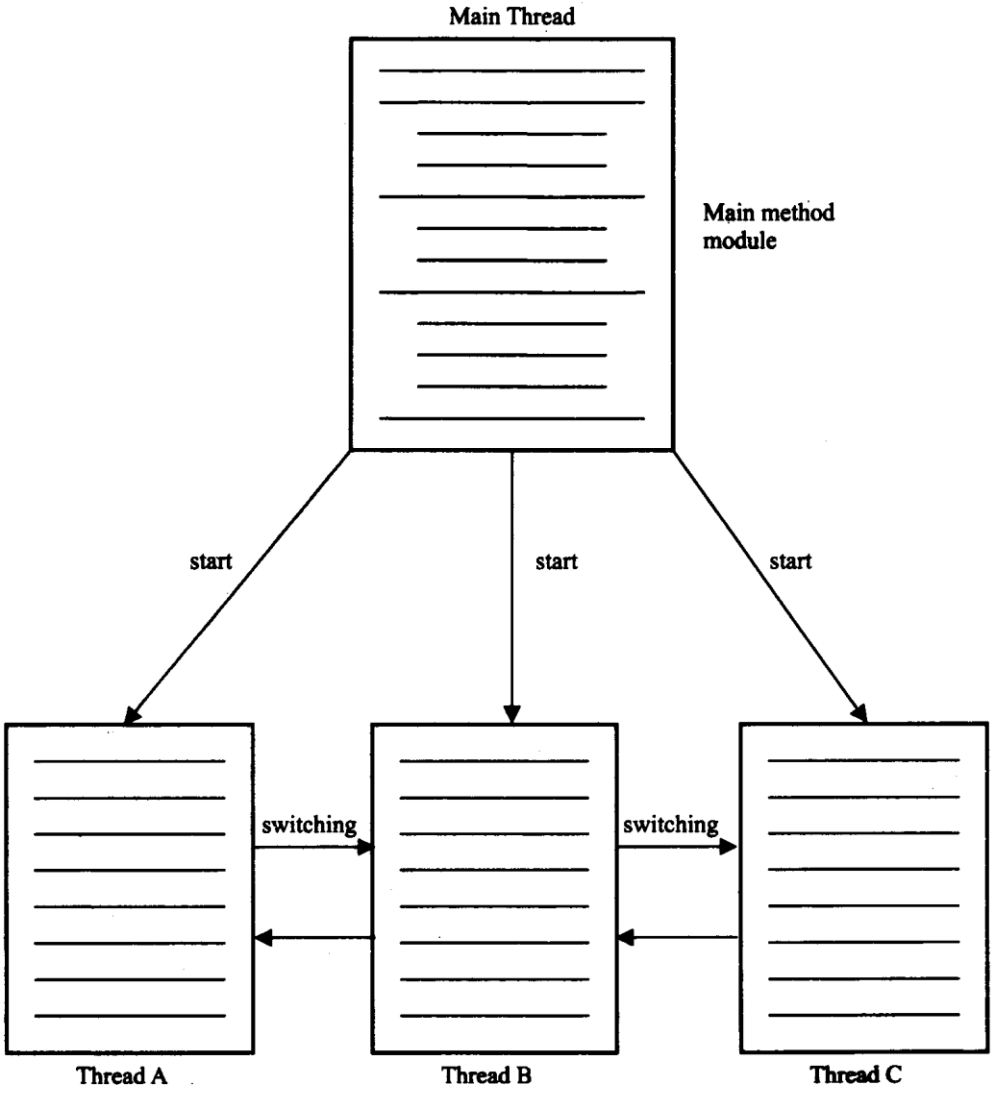
- We can execute several programs simultaneously. This ability is known as multitasking. In systems terminology, it is called **multithreading**.
- Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel.
- A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially.
- In fact, all main programs in our earlier examples can be called single-threaded programs. Every program will have at least one thread as shown in Fig. 12.1.

Fig. 12.1



- A unique property of Java is its support for multithreading. That is, Java enables us to use multiple flows of control in developing programs.
- Each flow of control may be thought of as a separate tiny program (or module) known as a thread that runs in parallel to others as shown in Fig. 12.2.
- A program that contains multiple flows of control is known as **multithreaded program**

Fig. 12.2



A Multithreaded program

Creating Threads

- Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method called run().

- The run() method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behaviour can be implemented. A typical run( ) would appear as follows:


```
public void run( )
{
    ..........
    .......... (statements for implementing thread)
    ..........
}
```

A new thread can be created in **two ways**.

1. **By creating a thread class:**

   Define a class that extends Thread class and override its run() method with the code required by the thread.

2. **By converting a class to a thread:**

   Define a class that implements Runnable interface. The Runnable interface has only one method, run ( ), that is to be defined in the method with the code to be executed by the thread.

Extending the Thread Class

We can make our class runnable as a thread by extending the class java.lang.Thread. This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the **Thread** class.

2. Implement the **run( )** method that is responsible for executing the sequence of code

that the thread will execute.

3. Create a thread object and call the **start()** method to initiate the thread execution.

Declaring the Class

The Thread class can be extended as follows:

class MyThread extends Thread

{ .....

..... .

..... .

}

Now we have a new type of thread MyThread.

Implementing the run() Method

The run() method has been inherited by the class **MyThread**. We have to override this method in order to implement the code to be executed by our thread. The basic implementation of run( ) will look like this:

public void run( )

{ .....

. . . . . / / Thread code here .....

}

When we start the new thread, Java calls the thread's run ( ) method, so it is the run ( ) where

all the action takes place.

Starting. New Thread

To actually create and run an instance of our thread class, we must write the following:

MyThread aThread = new MyThread( );

aThread. start ( ); / / invokes run() method

- The **first line** instantiates a new object of class MyThread. Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a newborn state.
- The **second line** calls the start ( ) method causing the thread to move into the runnable state.

An Example of Using the Thread Class

```
Class A extends Thread
{
    public void run( )
    {
        for(int i=l; i<=5;i++)
        {
```

```
                System.out.println("\tFrom Thread A : i - " + i);
            }
        System.out.println("Exit form A ");
        }
}
class B extends Thread
{
        public void run( )
        {
            for(int j=l; j<=5; j++)
            {
                System.out.println("\tFrom Thread B : j = " + j);
            }
            System.out.println("Exit from B ");
        }
}
class C extends Thread
{
        public void run( )
        {
            for(int k=l; k<=5; k++)
            {
                System.out.println("\tFrom Thread C : k = " + k);
            }
            System.out.println("Exit from C ");
        }
}
class ThreadTest
{
```

```
    public static void main (String args[ ])
    {
        new A ( ).start ( );
        new B ( ).start ( );
        new C ( ). start ( );
    }
}
```

OUTPUT:

```
From Thread A : i = 1
From. Thread A : i = 2
From Thread B : j = 1
.From Thread B : j = 2
From Thread C :k= 1-
From Thread C : k = 2
From Thread A : i = 3
From Thread A : i - 4
From Thread B : j = 3
From Thread B : j = 4
From Thread C : k = 3
From Thread C : k = 4
From Thread A : i = 5
Exit from A
From Thread B : j = 5
Exit from B
From Thread C : k = 5
Exit from C
```

STOPPING AND BLOCKING A THREAD

Stopping a Thread

We want to stop a thread from running further, we may do so by calling its stop( ) method, like:

aThread.stop( );

This statement causes the thread to move to the **dead state**. A thread will also move to the

dead state automatically when it reaches the end of its method. The **stop()** method maybe used when the premature death of a thread is desired.

Blocking a Thread

- A thread can also be temporarily suspended or blocked from entering into the runnable and

subsequently running state by using either of the following thread methods:

sleep ( ) / / blocked for a specified time

suspend ( ) / / blocked until further orders

wait ( ) / / blocked until certain condition occurs

- These methods cause the thread to go into the blocked (or not runnable) state. The thread will return to the runnable state when the specified time is elapsed in the case of sleep(), the resume( ) method is invoked in the case of suspend(), and the notify() method is called in the case of wait ( ).

LIFE CYCLE OF A THREAD

The life time of a thread, there are many states it can enter. They include:

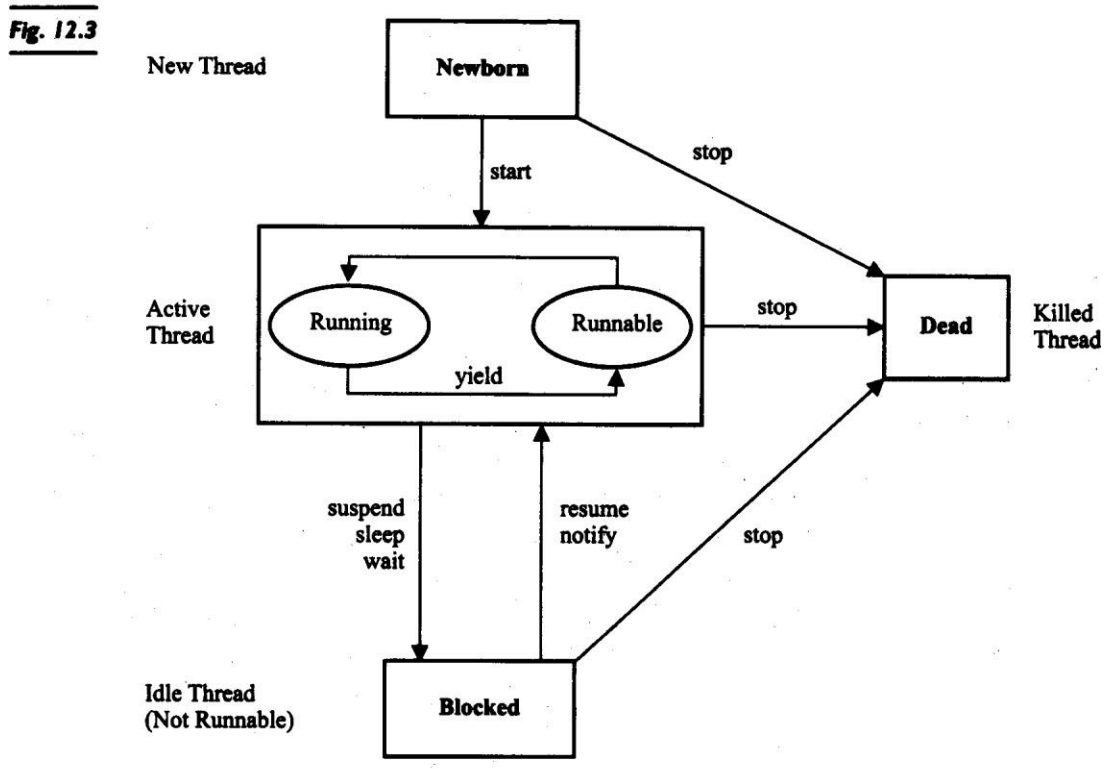
1. Newborn state

2. Runnable state

3. Running state

4. Blocked state ..

5. Dead state

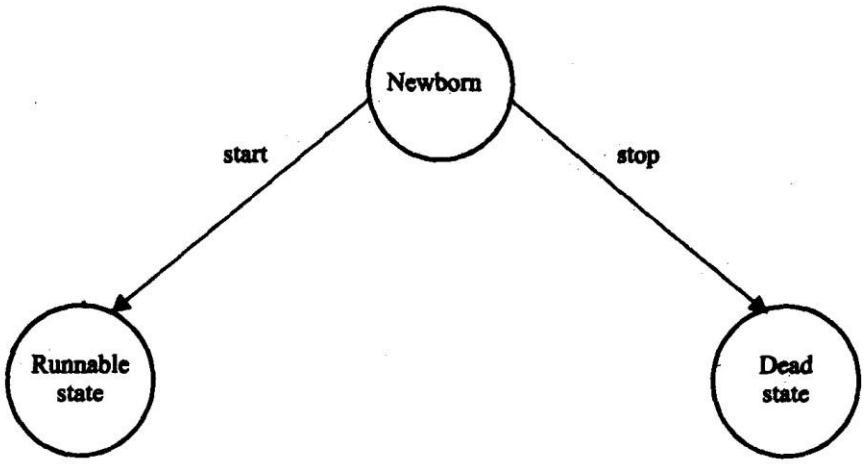


State transition diagram of a thread

Newborn State

- We create a thread object, the thread is born and is said to be in new born state. The thread
- is not yet scheduled for running. At this state, we can do onIy one of the following things with it:
  - Schedule it for running using start( ) method.
  - Kill it using stop( ) method.

Fig. 12.4

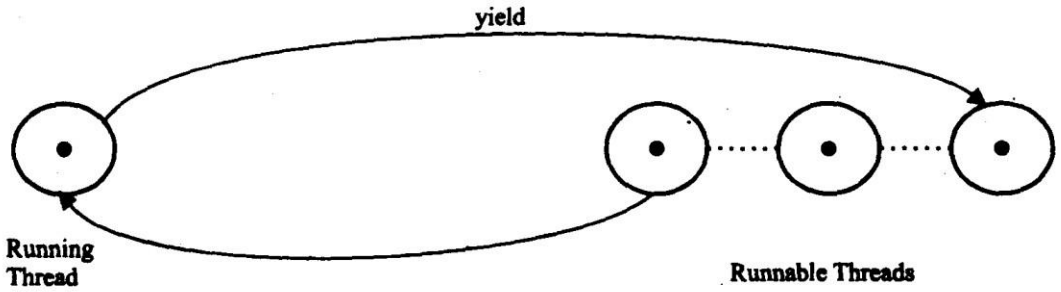


Scheduling a newborn thread

Runnable State

- The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.
- That is, the thread has joined the queue of threads that are waiting for execution.
- If we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using the **yield( )** method

Fig. 12.5



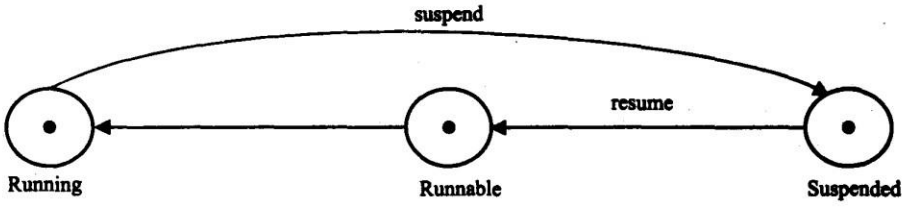
Relinquishing control using yield( ) method

Running State

- Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it· is preempted by a higher priority thread.
- A running thread may relinquish its control in one of the following situations

1. It has been suspended using **suspend( ) method. A suspended thread can be revived by using the resume( ) method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.**

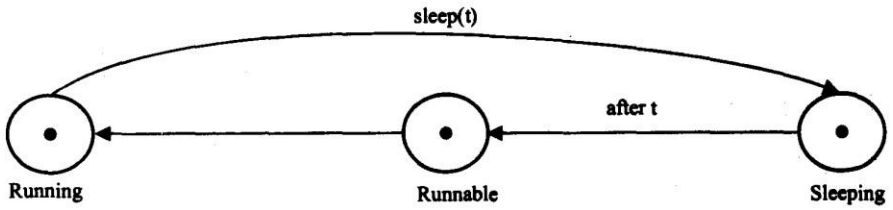
Fig. 12.6



Relinquishing control using suspend( ) method

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep(*time*)** where *time* is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.

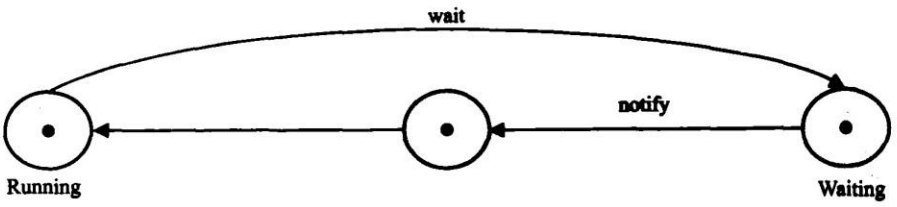
Fig. 12.7



Relinquishing control using sleep( ) method

3. It has been told to wait until some event occurs. This is done using the **wait( ) method.** The thread can be scheduled to run again using the **notify( ) method.**

Fig. 12.8



Relinquishing control using wait( ) method

Blocked State

- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.
- A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

DeadState

- Every thread has a life cycle. A running thread ends its life when it has completed executing its run{) method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it.
- A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.

USING THREAD METHODS

Use of yield ( ), sleep() and stop( ) methods.

```
class A extends Thread
{
        public void run( )
        {
                for(int i=l; i<=5; i++)
                {
                        if (i== 1) yield ( );
                        System.out.prinln("\tFrom Thread A : i =" +i);
                }
                System.out.println("Exit from A ");
        }
}
class B extends Thread
{
        public void run( )
        {
                for(int j=l; j<-5; j++)
```

```
                {
                        System.out.println("\tFromThread B : j =" +j);
                if (j==3) .top (.) ;
                }
                System.out.println("Exit from a");
        }
}
class C extends Thread
{
        public void run( )
        {
                for(int k=l; k<=5; k++)
                {
                System.out.println("\tFrom Thread C : k = " +k);
                if (k==l)
                try
                {
                        sleep(1000);
                }
                catch (Exception e)
                }
        }
}
System.out.println(~Exit from C ");
}
}
class ThreadMethods
{
        public static void main (String args[ ])
```

```
    {
        A threadA = new A( ) ;
        a threadB = new a( );
        C threadC = new C( ) ;
        System.out.println("Start thread A");
        threadA. start ( );
        System.out.println("Start thread air);
        threadB.start( );
        System.out.println("Start thread C·);
        threadC.start( ) ;
        System.out.println("End of main thread");
    }
}
```

OUTPUT

```
Start thread A
Start thread B
Start thread C
From Thread B: j = 1
From Thread B: j = 2
From Thread A: i = 1
From Thread A: i = 2
End of main thread
From Thread C: k = 1
From Thread B: j = 3
From Thread A: i = 3
From Thread A: i = 4
From Thread A: i = 5
Exit from A
From Thread C: k = 2
```

From Thread C: k = 3

From Thread C: k = 4

From Thread C: k = 5

Exit from C

THREAD EXCEPTIONS

- Java run system will throw **IllegalThreadStateException** whenever we attempt to invoke a method that a thread cannot handle in the given state.
- For example, a sleeping thread cannot deal with the resume( ) method because a sleeping thread cannot receive any instructions. The same is true with 'the suspend() method when it is used on it blocked (Not Runnable) thread.

```
catch (ThreadDeath e)
{ .....
.....                    // Killed thread
}
catch (InterruptedException e)
{ .....
.....            // Cannot handle it in the current state
}
catch (IllegalArgumentException e)
{ .....
.....            //Illegal method argument
}
catch (Exception e)
{
.....                    // Any other
.....
}
```

THREAD PRIORITY

- In Java, each thread is assigned a priority, it is scheduled for running. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a **first-come, first-serve basis.**

```
ThreadName.setPriority(intNumber);
```

The Thread class defines several.priority constants:

MIN PRIORITY = 1

NORM_PRIORITY = 5

MAXPRIORITY = 10

```
class A extends Thread
{
        public void run( )
        {
                System.out.println(“threadA started");
                for(int =-l; i<=4; i++)
                {
                        System.out.println("\tFrom Thread A : i - " +i);
                }
        System.out.println("Exit from A");
        }
}
class B extends Thread
{
        public void run( )
        {
                System.out.println("threadB started");
```

```
                for(i~t j=l; j<=4; j++)
                {
                        System.out.println("\tFrom Thread B : j = " +j);
                }
                System.out.println("Exit from B ");
        }
}
class C extends Thread
{
        public void run()
        {
                System.out.println("threadC started");
                for(int k=l; k<=4; k++)
                {
                        System.out.println("\tFrom Thread C : k.." +k);
                }
                System.out.println("Exit from C ");
        }
}

class ThreadPriority
{
        public static void main (String args[ ])
        {
                A threadA =new A( );
                B threadB = new B( );
                C threadC = new C( );
```

```
                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.set Priority (threadA.getPriority( )+1);
                threadA.setPriority(Thread.MIN_PRIORITY);
                System.out.println("Start thread A");
                threadA.start( );
                System.out.printlri("Start thread B");
                threadB.start( );
                System.out.println("Start thread C”);
                threadC.start.( );
                System.out.println("End of main thread-);
        }
}
```

Output:

```
Start  thread  A
Start  thread  B
Start  thread  C
threadB started
From Thread B : j. = 1
From Thread B : j =. 2
threadC started
FroIl,l Thread C : k = 1
From Thread C : k = 2
From Thread C : k = 3
From Thread C : k = 4
Exit from C
End of main thread
From Thread B : j = 3
From Thread B : j 7  4
Exit from B
threadA started
From Thread A : i = 1
From Thread A : i = 2
From Thread A : i = 3
From Thread A : i = 4
Exit from A
```

UNIT - V

MANAGING ERRORS, EXCEPTION AND APPLET PROGRAMMING

INTRODUCTION

- A mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that can make a program go wrong.
- An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash.

TYPES OF ERRORS

Errors may broadly be classified into two categories:

• Compile-time errors

• Run-time errors

Compile-Time Errors

- All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as **compile-time errors.**
- Whenever the compiler displays an error, it will not create the .class file.
- It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Most of the compile-time errors are due to typing mistakes. The most, common problems are:

• Missing semicolons

• Missing (or mismatch of) brackets in classes and methods

• Misspelling of identifiers and keywords

• Missing double quotes in strings

• Use of undeclared variables

• Incompatible types in assignments / initialization

• Bad references to objects

• Use of = in place of = = operator

• And so on

Run-Time Errors

- A program may compile successfully creating the .class file but may not run properly.
- Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overt1ow.
- Most common run-time errors are

• Dividing an integer by zero

• Accessing an element that is out of the bounds_of an array.

• Trying to store a value into an array of an incompatible class or type.

• Trying to cast an instance of a class to one of its subclasses.

• Passing a parameter that is not in a valid range or value for a method.

• Trying to illegally change the state of a thread.

• Attempting to use a negative size for an array.

• Using a null object reference as a legitimate object reference to access a method or a variable.

• Converting invalid string to a number.

• Accessing a character that is out of bounds of a string.

• And many more

Errors vs Exceptions

- **Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.
- **Exceptions** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.

EXCEPTIONS

- An Exception is a condition that is caused by a run-time error in the program.
- 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).

- 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 then display an appropriate message for taking corrective .actions. This task is known as exception handling.
- The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

  1. Find the problem (Hit the exception).
  2. Inform that an error has occurred (Throw the exception)
  3. Receive the error information (Catch the exception)
  4. Take corrective actions (Handle the exception)

The error handling code basically consists of **two segments**, **one** to detect errors and to throw exceptions and the **other** to catch exceptions and to take appropriate actions.

Common Java Exceptions

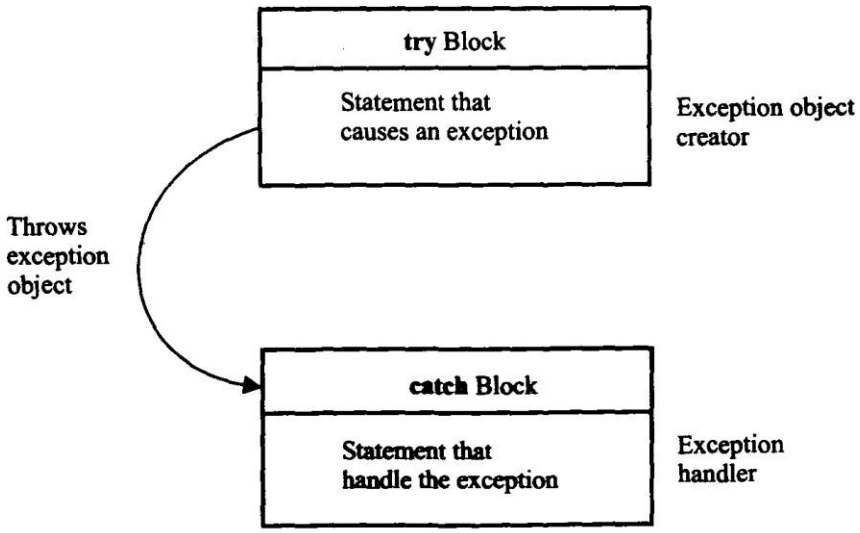
Exception Type	**Cause of Exception**
ArithmeticException	Caused by math errors such as division by zero
ArraylndexOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullpointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverflowException	Caused when the system runs out of stack space

StringIndexOutOfBoundsException	Caused when a program attempts to access a Non existent character position in a string

SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of exception handling are *throwing* an exception and *catching.*

Fig. 13.1



Exception handling mechanism

- The **try block** can have one or more statements that could generate an exception. If anyone 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.
- The **catch block** too can have one or more statements that are necessary to process the exception.

```
class Error3
{
        public static void main (String args[ ] )
        {
                int a =  10;
                int b =  5;
                int c =  5;
                int x, y ;
                try
                {
                        x = a / (b-c) ; // Exception here
                }.
                catch  (ArithmeticException e)
                {
                        System.out.println("U Division by zero");
                }
```

```
            y = a / (b+c) ;
            System.out.println(uy = U + y);
        }
}
```

Displays the following output:

Division by zero.
y . = 1

MULTIPLE CATCH STATEMENTS

- An exception in a try block is generated, the Java treats the multiple catch statements like cases in a switch statement.
- The first statement whose parameter matches with the exception object will be executed, and the remaining statements will skipped.
- Note that Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

```
class Error4
{
        public static void main (String args[ ] )
        {
                int a[ ] = {5,10};
                int b = 5;
                try
                {
                        int x = a[2] / b - a[l];
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Division by zero");
                }
                catch (ArraylndexOutOfBoundsException e)
                {
                        System.out.println("Array index error");
                }
                catch(ArrayStoreException e)
                {
                        System.out.println("Wrong data type");
                }
                int y = a[l] / a [0] ;
                System.out.println("y = "  +  y);
        }
}
```

uses a chain of catch blocks and, when run, produces the following output:
Array index error
y = 2

USING finally STATEMENT

- finally block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block.

- A finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. As a result, we can use it to perform certain house-keeping operations such as closing files and releasing system resources.

Syntax for finally Statement

```
try
{
      //Statements that may cause an exception
 }
 catch
{
     //Handling exception
}
 finally
 {
    //Statements to be executed
}
```

Example Program

a finally block as shown below:

```
finally
{
        int y = a[l]/a[O];
        System.out.println("y = " +y);
}
```

This will produce the same output.

THROWING OUR OWN EXCEPTIONS

There may be times when we would like to throw our own exceptions. We can do this by using the keyword throw as follows:

```
throw new Throwable_subclass;
```

Examples:

throw new ArithmeticException( );

throw new NumberFormatException( );

USING EXCEPTIONS FOR DEBUGGING

- The exception-handling mechanism can be used to hide errors from rest of the program. It is possible that the programmers may misuse this technique for hiding errors rather than debugging the code.
- Exception handling mechanism may be effectively used to locate the type and place of errors. Once we identify the errors, we must try to find out why these errors occur before we cover them up with exception handlers.

APPLET PRORAMMING

INTRDUOCTION

- Applets are small Java programs that are primarily used in Internet computing. They can be transported over the Internet from one computer to another and run using the Applet Viewer.

- Web browser that supports Java. An applet, like any application program, can do many things for us.
- It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation, and play interactive games.

- Java has revolutionized the way the Internet users retrieve and use documents on the world wide network. Java has enabled them to create and use fully interactive multimedia Web documents.
- A web page can now contain not only a simple text or a static image but also a Java applet run, can produce graphics, sounds and moving images.

- Java applets therefore have begun to make a significant impact on the World Wide Web.

Local and Remote Applets

- We can embed applets into Web pages in **two ways**.

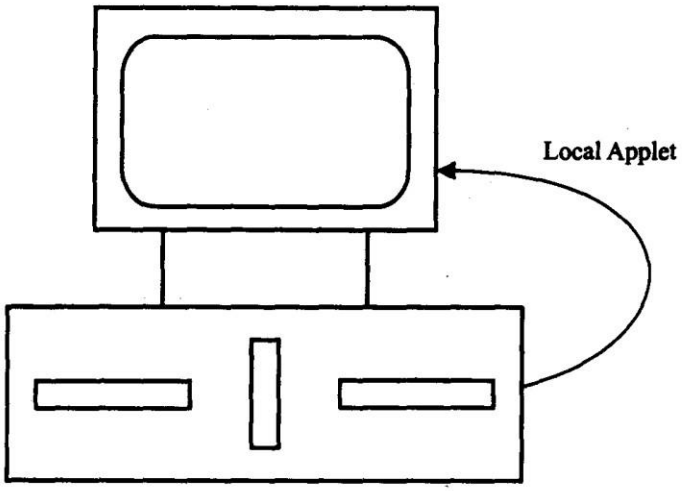
- **One**, we can write our own applets and embed them into Web pages. **Second**, we can download an applet from a remote computer system and then embed it into a Web page.

- An applet developed locally and stored in a local system is known as a local applet. When a Web page is trying to find a local applet, it does not need to use the Internet (see Fig.).

- A remote applet is developed by someone else Uid stored on a remote computer connected to the Internet. If our system is connected to the Internet, we can download the remote applet onto our system via the Internet and run it (see Fig.).

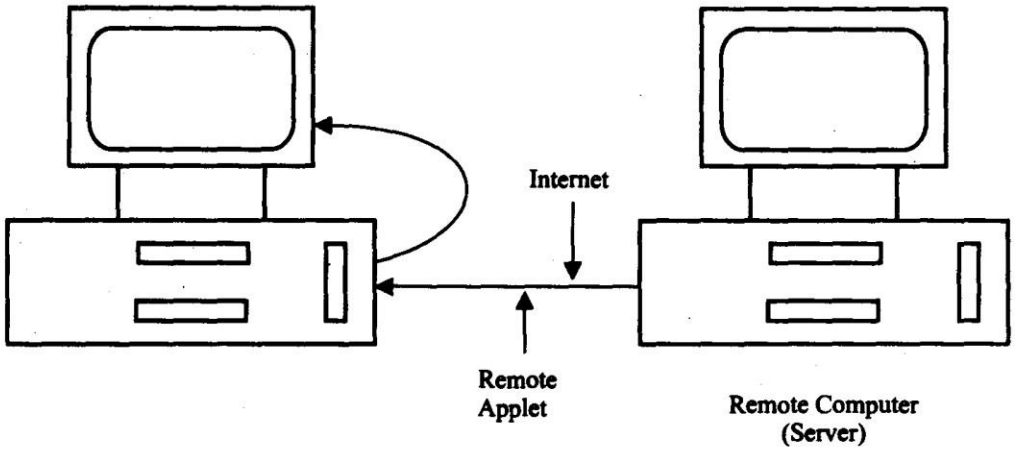
- In order to locate and load a remote applet, we must know the applet's address on the Web.
- This address is known as Uniform Resource Locator (URL) and must be specified in the applet's HTML document as the value of the CODEBASE attribute.

Fig. 14.1



Loading local applets

Fig. 14.2



Loading a remote applet

PREPARING TO WRITE APPLETS

We have been creating simple Java application programs with a single main( ) method that created objects, set instance variables and ran methods. Here, we will be creating applets exclusively and therefore we will need to know

- When to use applets,
- How an applets works,
- What sort of features an applet has, and
- Where to start when we first create our own applets.

The steps involved in developing and testing an applet are:

1. Building an applet code (.java file)
2. Creating an executable applet (.class file)
3. Designing a Web page using HTMLtags
4. Preparing <APPLET> tag
5. Incorpora~ing <APPLET>tag into the Web page
6. Creating HTMLfile
7. Testing the applet code.

BUILDING APPLET CODE

- It is essential that our applet code uses the services of two classes, namely, Applet and Graphics from the Java class library.

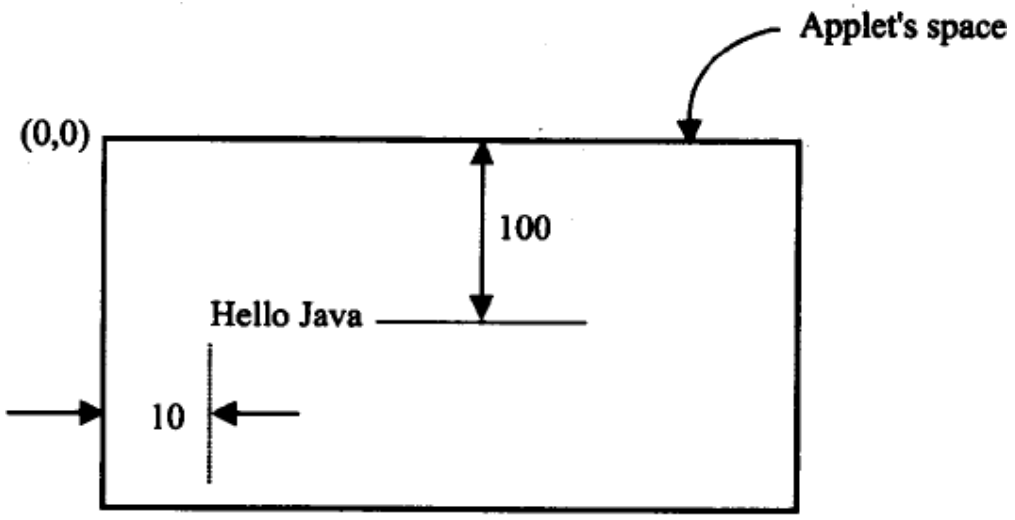
- The Applet class is contained in the java.applet .package provides life and behaviour to the applet through its methods such as **init( ), start( ) and paint( ).**

- Java calls the **main( )** method directly to initiate the execution of the program, when an applet is loaded, Java automatically calls a series of Applet class methods for **starting, running, and stopping** the applet code. The Applet class therefore maintains the lifecycle of all applet.

- The paint( ) method of the Applet class, when it is called, actually displays the result of the applet code on the screen. The output may be text, graphics, or sound. The paint( ) method, which requires a Graphics object as an argument, is defined as follows:

```
public void paint(Graphics g)
```

```
import java.awt.*
import java.applet.*;
public class HelloJava extends Applet
{
        public void paint (Graphics g)
        {
                g.drawString("Hello Java", 10, 100);
        }
}
```

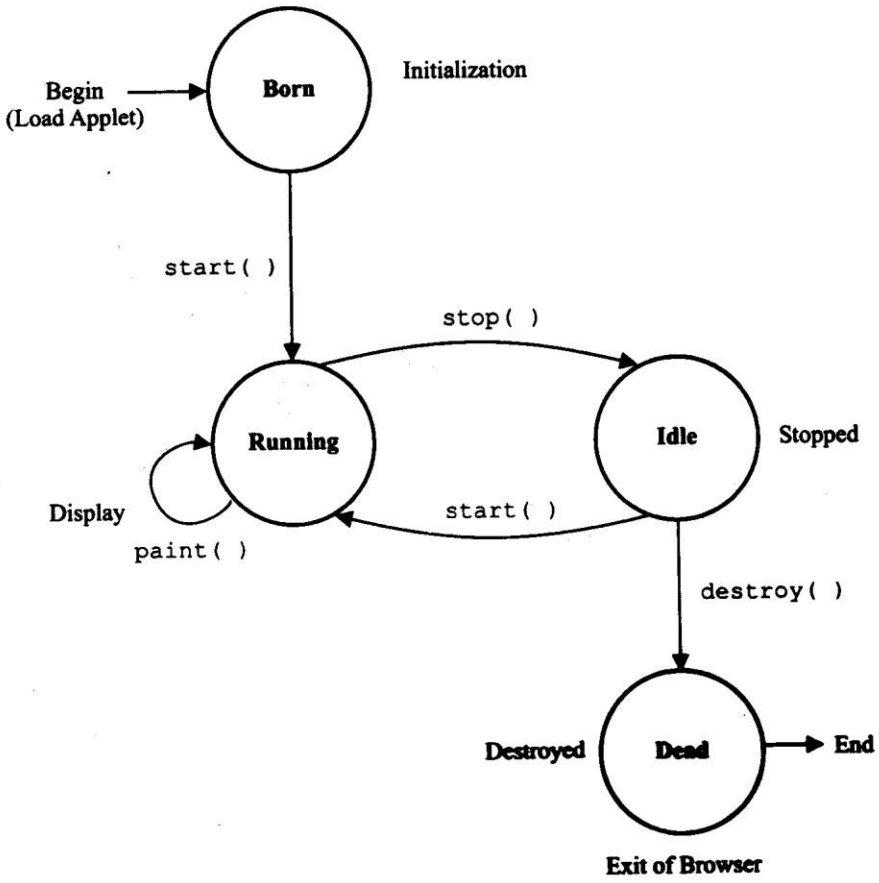


Output of Program

APPLET LIFE CYCLE

- Every Java applet inherits a set of default behaviours from the Applet class. As a result, when an applet is loaded, it undergoes a series of changes in its state as shown in Fig.
- The applet states include:
  - Born or initialization state
  - Running state
  - Idle state
  - Dead or destroyed state

Fig. 14.5



An applet's state transition diagram

Initialization State

- Applet enters the **initialization state** when it is first loaded. This is achieved by calling the init() method of Applet Class. The applet is born. At this stage, we may do the following, if required.

• Create objects needed by the applet

• Set up initial values

• Load images or fonts

• Set up colors

```
public void init( )
{ .....
..... (Action)
..... .
}
```

Running State

- Applet enters the running state when the system calls the start( ) method of Applet Class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in "stopped" (idle) state.
- For example, we may leave the Web page containing the applet temporarily to another page and return back to the page. This again starts the applet running.

```
public void start ( )
{ .....
. . . . . (Action) .....
}
```

Idle or Stopped State

- An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling the stop ( ) method explicitly.
- If we use a thread to run the applet, then we must use stop( ) method to terminate the thread. We can achieve this by overriding the stop( ) method.

```
public void stop( )
```

{ .....

..... (Action)

. ... '.....

}

Dead State

- An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the destroy() method when we quit the browser. Like initialization, destroying stage occurs only once in the applet's life cycle.
- we may override the destroy( ) method to clean up these resources.

public void destroy( )

{ .....

..... (Action) ..... .

}

Display State

Applet .moves to the display state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into the running state. The paint( ) method is called to accomplish this task. Almost every applet will have a paint( ) method.

public void paint (Graphics g}

{ .....

..... (Display statements)

..... .

}

CREATING AN EXECUTABLE APPLET

Executable applet is nothing but the .class file of the applet. which is obtained by compiling the source code of the applet. Compiling an applet is exactly the same as compiling an application.

Here are the steps required for compiling the HelloJava applet.

I. Move to the directory containing the source code and type the following command:
javac HelloJava.java
2. The compiled output file called HelloJava.class is placed in the same directory as the source.
3. If any error message is received, then we must check for errors, correct them and compile the applet again.

DESIGNING A WEB PAGE

- A Web page is basically made up of text and HTML tags that can be interpreted by a Web browser or an applet viewer. Like Java source code, it can be prepared using any ASCII text editor.
- A Web page is also known as HTML page or HTML document. Web pages are stored using a file extension .html such as MyApplet.html. Such files are referred to as HTML files.
- HTML files should be stored in the same directory as the compiled code of the applets. As pointed out earlier, Web pages include both text that we want to display and HTML tags (commands) to Web browsers.
- A Web page is marked by an opening <HTML> tag and a closing </HTML> and is divided into the following three major sections:

1. Comment section (Optional)
2. Head section (Optional)
3. Body section

Comment Section

- This section contains comments about the Web page. It is important to include comments that tell us what is going on in the Web page. A comment line begins with a
  <! and ends with a >.
- Web browsers will ignore the text enclosed between them. Although comments are important, they should be kept to a minimum as they will be downloaded along with the applet. Note that comments are optional and can be included anywhere in the Web page.

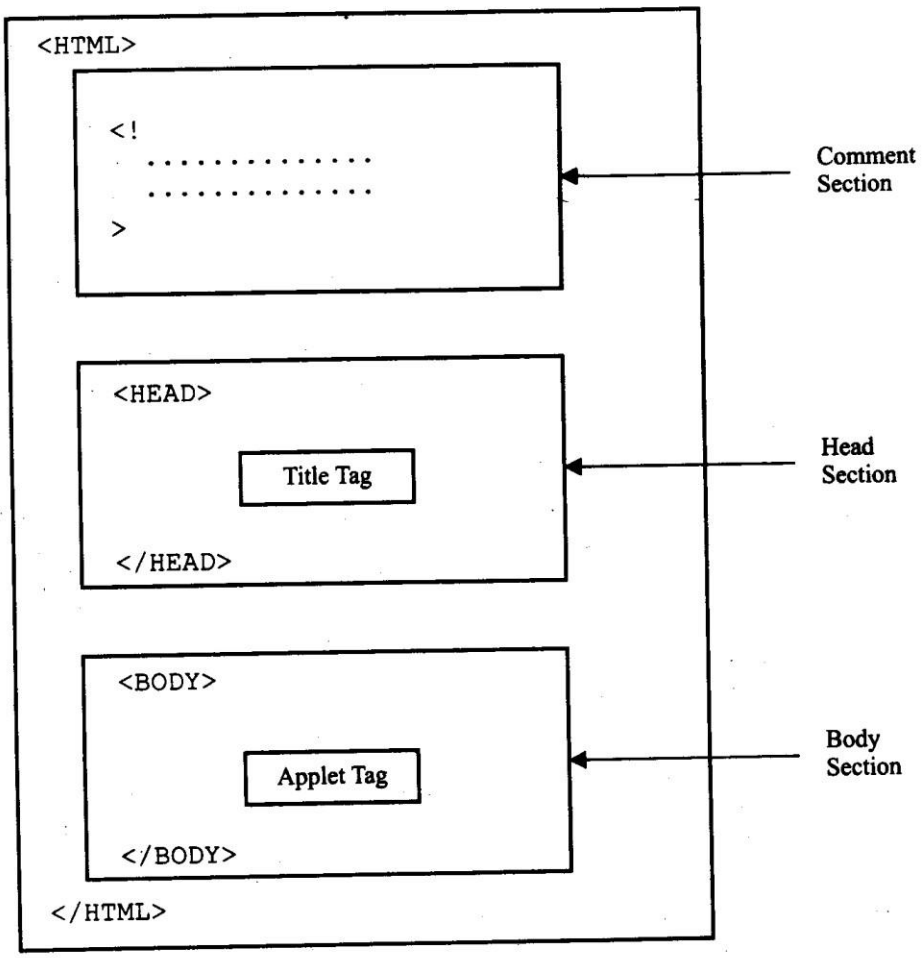


Fig. 14.6 A Web page template

Head Section

The head section is defined with a starting <HEAD> tag and a closing </HEAD> tag. This section contains a title for the Web page as shown below:

```
<HEAD>
<TITL~> Welcome to Java Applets </TITLE>
</HEAD>
```

The text enclosed in the tags <TITLE>and </TITLE> will appear in the title bar of the Web browser when it displays the page.

The head section is also optional.

Body Section

We call this as body section because this section contains the entire information about the Web page and its behaviour. We can set up many options to indicate how our page must appear on the screen (like colour, location, sound, etc. ,).

Shown below is a simple body section:

<BODY>
<CENTER>
<Hl> Welcome' to the World of Applets </Hl>
</CENTER>

<APPLET ... >
</APPLET>
</BODY>

The body shown above contains instructions to display the message
Welcome to the World of Applets

APPLET TAG

The <APPLET ... >tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires ..
The ellipsis in the tag <APPLET .. ,> indicates that it contains certain attributes that must specified. The <APPLET> tag given below specifies the minimum requirements to place the HeIloJava applet on a Web page:

<APPLET CODE = helloJava.class WIDTH = 400 HEIGHT = 200 >
</APPLET >

This HTML code tells the browser to load the compiled Java applet HeIloJava.class, which is in the same directory as this HTML file. And also specifies the display area for the applet output as 400 pixels width and 200 pixels height.

ADDING APPLET TO HTML FILE

Now we can put together the various components of the Web page and create a file known as HTML file. Insert the <APPLET> tag in the page at the place where the output of the applet must appear. Following is the content of the HTML file that is embedded with the <APPLET>
tag of our HelloJava applet.

```
<HTML>
    <! This page includes a welcome title in the title bar and also displays a welcome message. Then it specifies the applet to be loaded and executed. >
    <HEAD>
        <TITLE> Welcome to Java Applets </TITLE>
    </HEAD>
    <BODY>
        <CENTER>
            <Hl> Welcome to the World of Applets </Hl>
        </CENTER>
        <BR>
        <CENTER>
            <APPLET CODE = helloJava.class WIDTH = 400 HEIGHT = 200 >
            </ APPLET >
        </CENTER>
    </BODY>
</HTML>
```

RUNNING THE APPLET

- Now that we have created applet files as well as the HTML file containing the applet, we must have the following files in our current directory:

  HelloJava.java

  HelloJava.class

  HelloJava.html

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

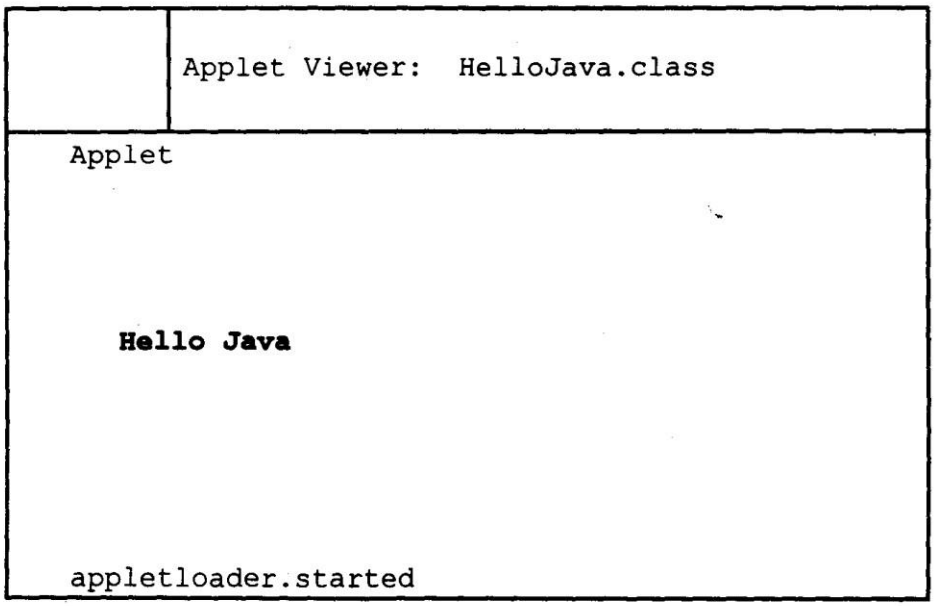
1. Java-enabled Web browser (such as HotJava or Netscape)

2. Java appletviewer

If we use a Java-enabled Web browser, we will be able to see the entire Web page containing the applet. If we use the appletviewer tool, we will only see the applet output.

```
appletviewer HelloJava.html
```

Fig. 14.7



Output of HelloJava applet by using appletviewer

MORE ABOUT APPLET TAG

- We have used the <APPLET> tag in its simplest form. In its simplest form, it merely creates a space of the required size and then displays the applet output in that space.
- The syntax of the <APPLET> tag is a little more complex and includes several attributes

<APPLET

[CODEBASE= codebase_URL]

CODE = AppletFileName.class

[ALT = alternate_text]

[NAME= applet_instance_name]

WIDTH = pixels

HEIGHT = pixels

[ALIGN = alignment]

[VSPACE = pixels]

[HSPACE = pixels]

>

[< PARAMNAME= namel VALUE = valuel>]

[< PARAMNAME= name2 VALUE = value2>]

.....

..... -....

[Text to be displayed in the absence of Java]

</ APPLET >

Attribute	Meaning
CODE=AppletFileName.class	Specifies the name of the applet class to be loaded. That is, the name of the already-compiled .cIass file in which the executable Java bytecode for the applet is stored.
CODEBASE=codebase URL (Optional)	Specifies the URL of the directory in which the applet resides. If the. applet resides in the same directory as the HTML file, then the CODEBASE attribute may be omitted entirely.
WIDTH-pixels HEIGHT=pixels	These attributes specify the width and height of the space on the HTMLpage that will be reserved lor the applet.
NAME=applet_instance_name (Optional)	A name lor the applet may optionally be specified so that other applets on the page may reler to this applet. This lacilitates interapplet communication.
ALIGN=alignment (Optional)	This optional attribute specifies where on the page the applet will appear. Possible

	values lor alignment are: TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABSMIDDLE, ABSBOTIOM, TEXTTOP, and BASELINE.
HSPACE=pixels (Optional)	Used only when ALIGN is set to LEFT or RIGHT, this .attribute specifies the amount of horizontal blank space the browser shouldleave surrounding the applet.
VSPACE=pixels (Optional)	Used only when some vertical alignment is specilied with the ALIGN attribute (TOP, BOTTOM, etc.,) VSPACE specilies the amount of vertical blank space the browser should leave surrounding the applet.
ALT=alternate_text (Optional)	Non_Java browsers will display this text where the applet would normally go. This attribute is optional.

PASSING PARAMETERS TO APPLETS

- We can supply user-defined parameters to an applet using <PARAM... > tags. Each <PARAM... > tag has a name attribute such as color, and a t1aIue attribute such as red. Inside the applet code, the applet can refer to that parameter byname to find its value.
- For example, We can change the colour of the text displayed to red by an applet by using a <PARAM... > tag as follows:

```
.<APPLET..... >
<PARAMNAME= "color" VALUE= "red">
</APPLET>
```

```
import j ava. awt.* ;
import java.applet.*;
public class HelloJavaParam extends Applet
{
        String str;
        public void init( )
        {
                str = getParameter("string"); / / Receiving parameter value
                if (str, == nill)
                str = "Java";.
                str = "Hello" + str; / / Using the value
        }
        public void paint (Graphics g)
        {
        g.drawString(str, 10, 100) ;
        }
}
```

ALIGNING THE DISPLAY

We can align the applet space using the ALIGN attribute. This attribute can have one of the **nine values:**

LEFT, RIGHT,

TOP, TEXTTOP,

MIDDLE, ABSMIDDLE,

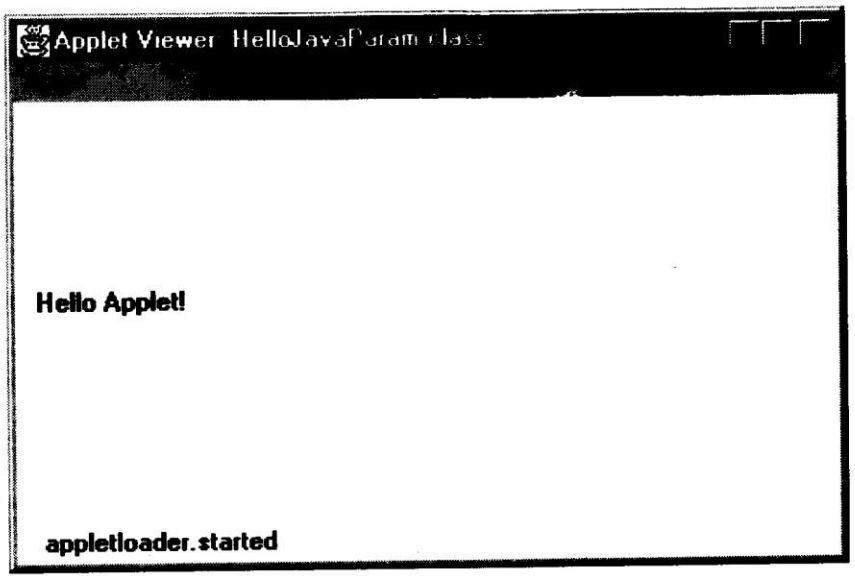
BASELINE, BOTI'OM,

ABSBOTTOM.

```
<HTML>
        <HEAD>
                <TITLE> Here is an applet </TITLE>
        </HEAD>
        <BODY>
                <APPLET CODE = HelloJava.class WIDTH = 400 HEIGHT = 200
                        ALIGN = RIGHT >
                </APPLET>
```

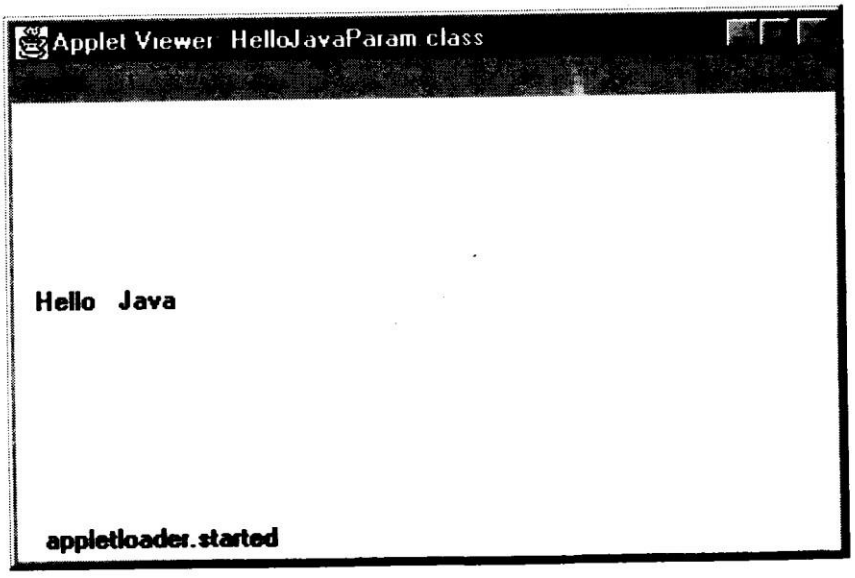
```
    </BODY>
</HTML>
```


Fig. 14.8



Displays of HelloJavaParam applet

Fig. 14.9



Output without PARAM tag

MORE ABOUT HTML TAGS

Tag	Function
<HTML> .•.•.• </HTML>	Signifies the beginning and end of a HTML file.
<HEAD> •.•.• </HEAD>	This tag may include details about the Web page. Usually contains <TITLE> tag within it.
<TITLE> •.••• </TITLE>	The text contained in it will appear in the title bar of the browser.
<BODY> .••.• </BODY>	This tag contains the main text of the Web page. It is the place where the <APPLET> tag is declared.
<Hl> .•..• </Hl>	Header tags. Used to display headings. <HI> creates the largest font .. header, while <H6> creates the smallest one .
<H6> •.•.• <H/6>	
<CENTER> .•. < / CENTER>	Places the text contained in it at the centre of the page.
<APPLET .•.• > <APPLET ... >	tag declares the applet details as its attributes.
<APPLET> ••• </APPLET>	May hold optionally user-defined parameters using <PARAM> Tags.
<PARAM •.•• >	Supplies user-defined parameters. The <PARAM> tag needs to be placed between the <APPLET> and </ APPLET> tags. We can use as many different <PARAM> tags as we wish for each applet.
<B> .... <B>	Text between these tags will be displayed in bold type.
 	Line break tag. This will skip a line. Does not have an end tag.
<P>	Para tag. This tag moves us to the next line and starts a paragraph of text. No end tag is necessary.
<IMG ., .>	This tag declares attributes of an image to be displayed.
<HR>	Draws a horizontal role.
<A ..•• > < / A>	Anchor tag used to add hyperlinks.

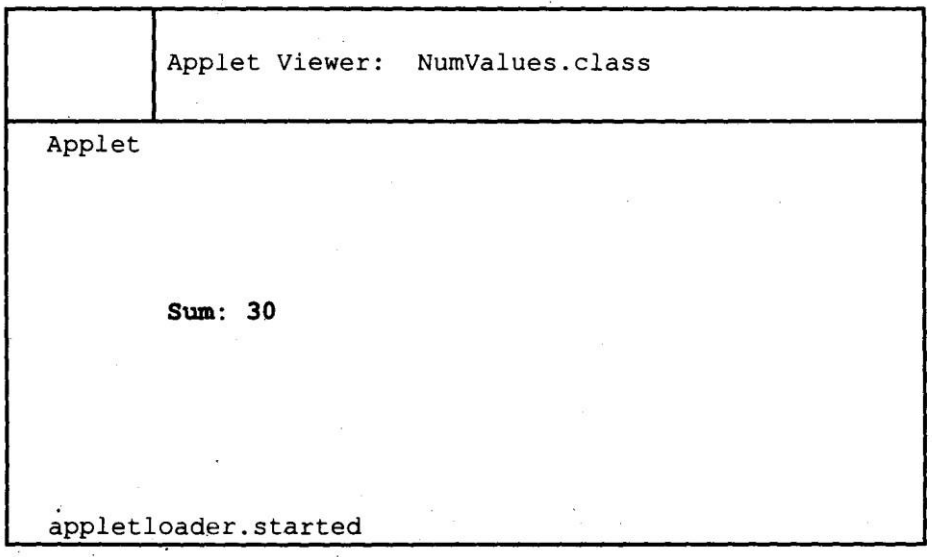
<FONT ••• > •.•• </FONT>	We can change the colour and size of the text that lies in between <FONT> and </FONT> tags ~ COWR and SIZE attributes in the tag <FONT ... >.
<! .••• >	Any text starting v.1th a <! mark and ending with a > mark is ignored by the Web browser. We may add comments here. A comment tag may be placed anywhere in a Webpage.

DISPLAYING NUMERICAL VALUES

In applets, we can display numerical values by first converting them into strings and the using the drawString( ) method of Graphics class. We can do this easily by calling the ValueOf( ) method of String class.

```
 import java.awt.*;
import java.applet.*;
public class NumValues extends Applet
{
        public void paint (Graphics g)
        {
                int valuel = 10;
                int value2 = 20;
                int sum = valuel + value2:
                String s = "sum: " + String.valueOf(sum);
                g.drawString(s, 100, 100) ;
        }
}
```

Fig 14.11



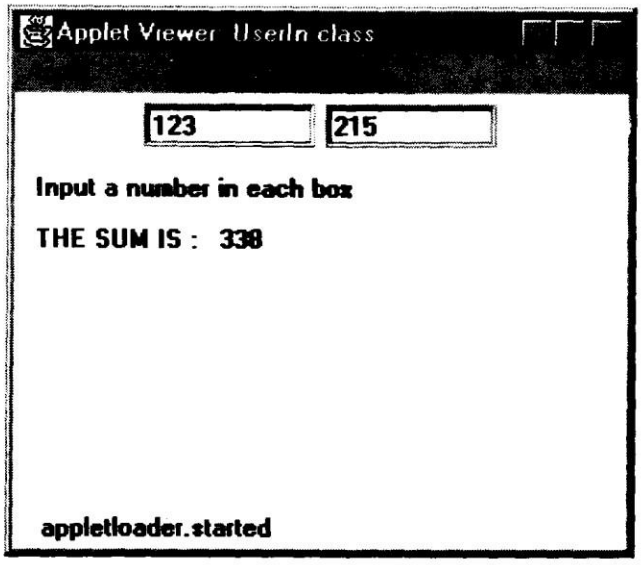
Output of Program 14.5

The applet Program 14.5 when run using the following HTML file displays the output as shown in Fig. 14.11.

```
import java.awt.*;
import java.applet.*;
public class Userln extends Applet
{
        TextField = textl, text2;
        public void init( )
        {
                textl = new TextField(B);
                text2 = new TextField(8);
                add (textl) ;
                add          (text2);
                textl.setText ("0") ;
                text2.setText ("0") ;
        }
        public void paint (Graphics g)
        {

                int x=O,y=O,z=O;
                String sl,s2,s;
                g.diawString(UInput a number in each box ",10,50);
                try
                {
                        sl = text1.getText( ) ;
                        x = Integer.parseInt(sl);
                        s2 = text2.getText( ) ;
                        y = Integer.parseInt(s2);
                }
                catch (Exception ex) { }
```

```
                z = x + y;
                s = String.valueOf(z);
                g.drawString(UTHE SUMIS: . ",10,75);
                g.drawString(s,100,75);
        }
        public boolean action(Event event, Object obj)
        {
                repaint ( ) ;
                return true;
        }
}

<html>
<app1et code = NumValues.class width = 300 height = 200 >
</applet>
</html>
```


Fig. 14.12



Interactive computing with applets

GRAPHICS PRORAMMING

THE GRAPHICS CLASS

- Java's Graphics class includes methods for drawing many different types of shapes, from simple lines to polygons to text in a variety of fonts. We have already seen how to display text using the paint( ) method and a Graphics object.

Table 15.1 Drawinl Methods of the Graphics Class

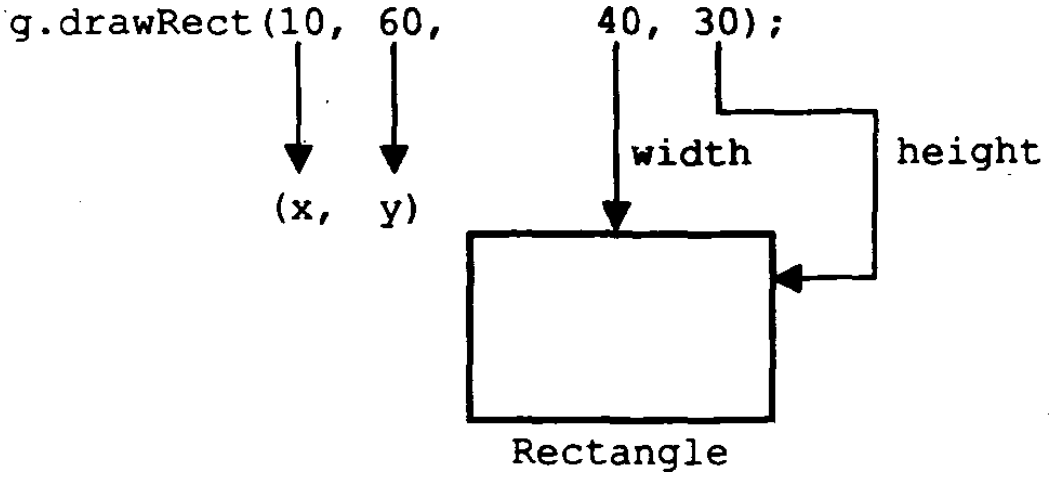
Method	Description
clearRect( )	Erases a rectangular area of the canvas.
copyArea( )	Copies a rectangular area of the canvas to another area.
drawArc( )	Draws a hollow arc.
drawLine( )	Draws a straight line.
drawOval( )	Draws a hollow oval.
drawPolygon( )	Draws a hollow polygon.
drawRect( )	Draws a hollow rectangle.
drawRoundRect( )	Draws a hollow rectangle with rounded comers.
drawString ( )	Displays a text string.
fillArc ( )	Draws a filled arc.
fillOval ( )	Draws a filled oval.
fill Polygon ( )	Draws a filled polygon.
fillRect ( )	Draws a filled rectangle.
fillRoundRect ( )	Draws a filled rectangle with rounded comers.
getColor( )	Retrieves the current drawing. colour.
getFont ( )	Retrieves the currently used font.
getFontMetrics( )	Retrieves information about the current font.
setColor ( )	Sets the drawing colour.
set Font ( )	Sets the font.

LINES AND RECTANGLES

- The simplest shape we can draw with the Graphics class is a line. The drawLine () method takes two pair of coordinates, (xl, yl) and (x2, y2) as arguments and draws a line between them.
- For example, the following statement draws a straight line from the coordinate point (10,10) to (50,50) :

g.drawLine(lO,lO, 50,50);

- we can draw a rectangle using the drawRect( ) method. This method takes four arguments. The first two represent the x and y coordinates of the top left corner of the rectangle, and the remaining two represent· the width and the height of the rectangle.
- For example, the statement



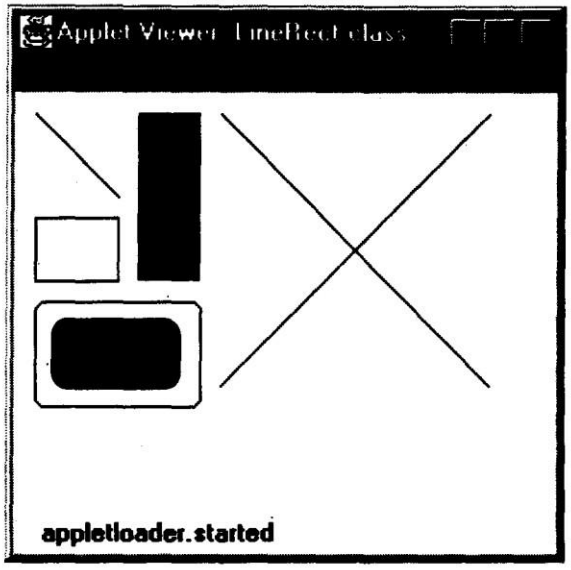

```
import java.awt.*;
import java.applet.*;
public class LineRect extends Applet
{
    public void paint {Graphics g)
    {
    g.drawLine{10, 10, 50, 50) ;
    g.drawRect{10, 60, 40, 30) ;
    g. fi11Rect (60, 10, 30, 80) ;
```

```
        g.drawRoundRect{10, 100, 80, 50, 10, 10) ;
        g.fillRoundRect{20,110, 60, 30, 5, 5) ;
        g. drawLine (1 00, 10, 230, 140) ;
        g.drawLine{100, 140, 230, 10) ;
        }
}
```

Program 15.2 LineRect.html file

```
<APPLET
      CODE = LineRect.class
      WIDTH = 250
      HEIGHT = 200>
</APPLET>
```

Fig. 15.2



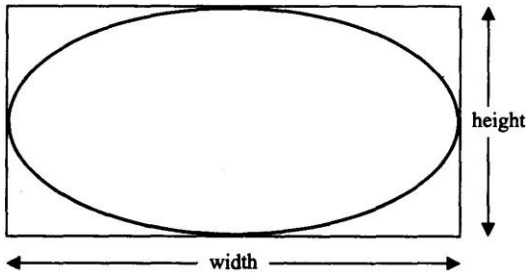
Output of LineRect applet

15.4 CIRCLES AND ELLIPSES

circles or ellipses

- The Graphics class does not have any method for circles or ellipses. the drawOval( ) method can be used to draw a circle or an ellipse.
- The drawOval( ) method takes four arguments: the first two represent the top left comer of the imaginary rectangle and the other two represent the width and height of the oval itself. Note that if the width and height are the same, the oval becomes a circle.
- Like rectangle methods, the drawOval( ) method draws outline of an oval, and the fillOval( ) method draws a solid oval. The code segment shown below draws a filled circle within an oval

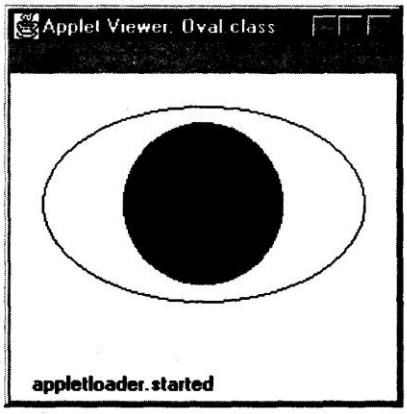
Fig. 15.3



Oval within an imaginary rectangle

```
public void paint(Graphics g)
{
   g.drawOval(20, 20, 200, 120);
   g.setColor(Color.green);
   g.fillOval(70, 30, 100, 100);    // This is a circle.
}
```

Fig. 15.4



A filled circle within an ellipse

We can draw an object using a color object as follows:

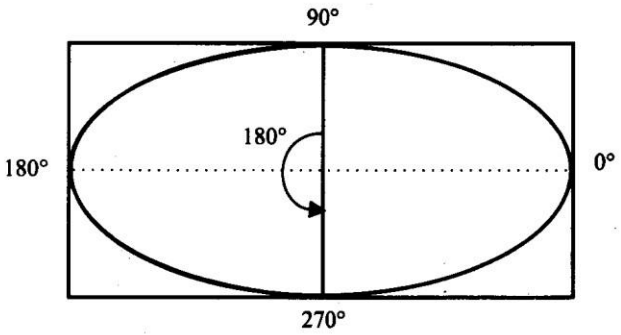
```
g.setColor(Color.green);
```

After setting the color, all drawing operations will occur in that color.

DRAWING ARCS

- The drawArc() designed to draw arcs takes six arguments. The first four are the same as the arguments for drawOval( ) method and the last two represent the starting angle of the arc and the number of degrees (sweep angle) around the arc.

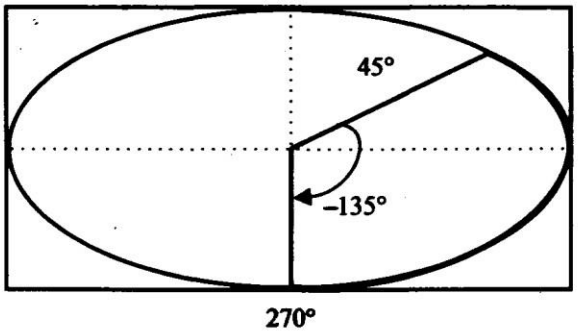
Fig. 15.5



Arc as a part of an oval

We can also draw an arc in backward direction by specifying the sweep angle as negative. For example, if the last argument is –135° and the starting angle is 45°, then the arc is drawn as shown in Fig. 15.6.

Fig. 15.6

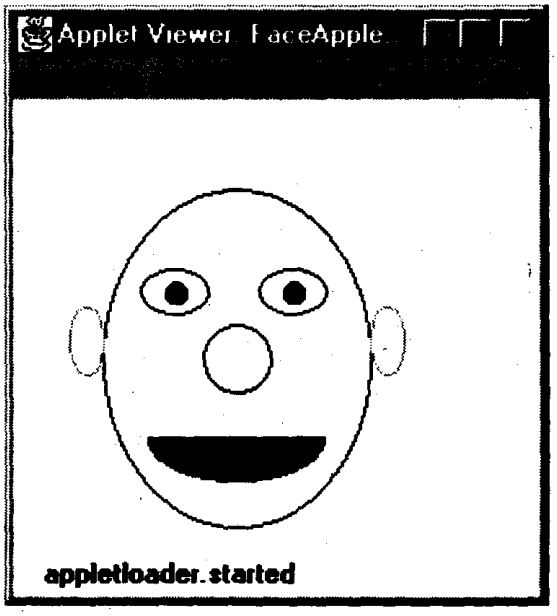


Drawing an arc in clockwise direction

```
import java.awt.*;
import java.applet.*;
pqblic class Face extends Applet
{
    public void paint (Graphics g)
    {
        g.drawOval(40, 40, 120, 150); // Head
```

```
        g.drawOva1 (57, 75, 30, 20); // Left eye
        g.drawOval (110, 75, 30, 20) ; // Right eye
        g. fillOval (68, 81, 10, 10) ; //Pupil (left)
        g.fillOval (121, 81, 10, 10) ; // Pupil (right)
        g.drawOval(85, 100, 30, 30) ; // Nose
        g.fillArc(60,125,80,40,180,180); // Mouth
        g.drawOval(25, 92, 15, 30) ; // Left ear
        g.drawOval(160, 92, 15, 30) ; // Right ear
    }
}
```

Fig. 15.7



Output of the Face applet

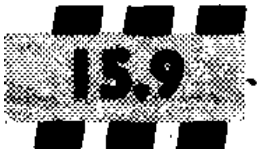
DRAWING POLYGONS

- Polygons are shapes with many sides.
- A polygon may be considered a set of lines connected together. The end of the first line is the beginning of the second line, the end of the second is the beginning ofthe third, and so on.
- This suggests that we can draw a polygon with n sides using the drawLine( ) method n times in succession. For example, the code given below will draw a polygon of three side
- We can draw polygons more conveniently using the drawPolygon( ) method of Graphics class. This method takes three arguments:

• An array of integers containing x coordinates

• An array of integers containing y coordinates

• An integer for the total number of points



DRAWING BAR CHARTS

Applets can be designed to displaybar charts, which are commonlyused in comparative analysis of data. The table below shows the annual turnover of a company during the period 1991 to 1994.These values maybe placed in a HTMLfile as PARAMattributes and then used in an applet

for displaying a bar chart.

Year	1991	1992	1993	1994
Turnover	110	150	100	170

```
import java.awt.*;
import java.applet.*;
public class BarChart extends Applet
{
        int n = 0;
        String label[ ];
        int value [ l;
        public void init( )
        {
        try
        {
                n = Integer~parselnt(getParameter(~columns"»;
                label = newString[n];
                value = new int[n];
                label  [0]  =  getparameter(·labell");
                label[l]  =  getParameter(·labe12");
                label[2]  =  getParameter(·labe13");
                labe~[3] = getParameter(ftlabe14");
                value [0] = Integer.parselnt (getparameter (“cl");
                value[l] = Integer.parselnt(getParameter(“c2");
                value[2] = Integer.parselnt(getParameter(“c3")
                value[3] = Integer.parseInt(getParameter("c4"))i
        }
        catch (NumberFormatException e). { }
        }
        public void paint (Graphics g)
        {
```

```
        for(int i = 0; i < ni i++)
        {
                g.setColor(Color.red)i
                g.drawString(label[i], 20, i*50+30)i
                g.fillRect(50,i*50+10,value[i],40);
        }
}
}
```

Program 15.' HTML file for runnln, the BarChart appIet

```
<HTML>
<APPLET
CODE = BarChart.class
WIDTH = 300
HEIGHT = 250>
<PARAM NAME = "columns" VALUE "" "4">
<PARAM NAME = "c1" VALUE "" "110">
<PARAM NAME = "c2" VALUE •.• "150">
<PARAM NAME = "c3" VALUE = "100">
<PARAM NAME "" "c4" VALUE = "170">
<PARAM NAME = "label1" VALUE •.• "91">
<PARAM.NAME = "labe12" .VALUE = "92"> .
<PARAM NAME = "labe13" VALUE = "93">
<PARAM NAME "" "labe14" VALUE - "94">
</APPLET>
</HTML>
```