

**GOVERNMENT OF TAMILNADU  
DIRECTORATE OF TECHNICAL EDUCATION  
CHENNAI – 600 025**

**STATE PROJECT COORDINATION UNIT**

**Diploma in Computer Engineering**

**Course Code: 1052**

**M – Scheme**

**e-TEXTBOOK  
on  
OBJECT ORIENTED PROGRAMMING WITH JAVA  
for  
IV Semester DCOMP**

**Convener for COMP Discipline:**

**Mrs.A.Ghousia Jabeen,**  
Principal,  
Thanthai Periyar E.V.Ramasamy Govt. Polytechnic College for Women,  
Vellore – 632 002

**Team Members for Object Oriented Programming With Java:**

**Mr.M.Jeyapal,**  
Lecturer/HOD incharge ,  
Department of Computer Engineering  
113, Government Polytechnic College,  
Aranthangi – 614616

**Mrs.N.Swarnalatha,**  
Lecturer (Sr.Gr.),  
280, A.D.J. Dharmambal Polytechnic College,  
Nagapattinam – 641 042

**Mrs.K.Valarmathi,**  
HOD incharge,  
Department of Computer Engineering  
225, S.Vellaichamy Nadar Polytechnic College,  
Virudhunagar – 626 001

**Validated By**

**Mr. V.G.Ravindhren**  
Lecturer(SG)/HOD incharge  
Seshayee Institute of Technology,  
Trichy- 620 010

**STATE BOARD OF TECHNICAL EDUCATION & TRAINING, TAMILNADU.**

**DIPLOMA IN COMPUTER ENGINEERING**

**M- SCHEME**

( to be implemented to the student Admitted from the Year 2015-2016 on wards)

Course Name : Diploma in Computer Engineering.

Subject Code : 35243

Semester :IV

Subject title : Object Oriented Programming with Java

**TEACHING & SCHEME OF EXAMINATION:**

No. of weeks per Semester 15 Weeks

Subject	Instructions		Examination			Duration
	Hours / Week	Hours / Semester	Internal Assessment	Board Examination	Total	
Object Oriented Programming with Java	6	90	25	75	100	3 Hrs

**UNITS AND ALLOCATION OF HOURS**

UNIT No.	TOPIC	No. of Hours
I	INTRODUCTION TO OOPS AND JAVA	15
II	CONTROL STRUCTURES, ARRAYS, AND VECTORS	13
III	STRINGS, CLASSES AND INTERFACES	18
IV	PACKAGES, APPLETS AND AWT CONTROLS	16
V	EXCEPTION HANDLING, MULTITHREADS AND I/O STREAMS	18
	TEST AND REVISION	10
	TOTAL	90

**Rationale:**

Today almost every branch of computer science is feeling presence of object - orientation. Object oriented technology is successfully incorporated in various fields of computer science. Since its arrival on the scene in 1995, the Java has been accepted as one of the primary programming language. This subject is designed to give you exposure to basic concepts of object - oriented technology. This subject will help in learning to write programs in Java using object - oriented paradigm. Approach in this subject is to take Java as a language that is used as a primary tool in many different areas of programming work.

**Objectives:**

On completion of the following units of syllabus contents, the students must be able to

- Know the paradigms of programming languages.
- Understand the concepts of Object Oriented Programming.
- State the benefits and applications of Object Oriented Programming.
- Know the history of development of Java.
- Comprehend the features and tokens of Java.
- Explain about the control structures used in Java.
- Use of Arrays and Vectors in Java Program.
- Demonstrate the use of string and String Buffers.
- Define Class with the attributes and methods.
- Understand the need for interfaces.
- Implement Interfaces in classes.
- Create packages.
- Write simple Applets.
- List the types of AWT Components and types of exceptions.
- Handle the errors using exceptions.
- Understand the concepts of multithreading.
- Develop multithreaded programs in Java.
- Define stream and list the types of streams.

**DETAILED SYLLABUS**

<b>UNIT I INTRODUCTION TO OOPS AND JAVA</b>		<b>15 HOURS</b>
1.1	<b>Introduction to OOPS:</b> Paradigms of Programming Languages - Basic concepts of Object Oriented Programming – Differences between Procedure Oriented Programming and Object Oriented programming - Objects and Classes – Data abstraction and Encapsulation, Inheritance, Polymorphism, Dynamic binding, Message communication – Benefits of OOP – Application of OOPs.	8 Hrs

1.2	<b>Java</b> : History – Java features – Java Environment – JDK – API.	2 Hrs
1.3	<b>Introduction to Java</b> : Types of java program – Creating and Executing a Java program – Java Tokens: Keywords, Character set, Identifiers, Literals, Separator – Java Virtual Machine (JVM) – Command Line Arguments – Comments in Java program	5 Hrs
<b>UNIT II CONTROL STRUCTURES, ARRAYS, AND VECTORS</b>		<b>13 HOURS</b>
2.1	Elements: Constants – Variables – Data types - Scope of variables – Type casting – Operators: Special operators – Expressions – Evaluation of Expressions	5 Hrs
2.2	Decision making and Branching: Simple if statement – if – else statement – Nesting if – else – else if Ladder – switch statement – Decision making and Looping: While loop – do – While loop - for loop – break – labeled loop – continue Statement.	5 Hrs
2.3	Arrays: One Dimensional Array – Creating an array – Array processing – Multidimensional Array – Vectors – ArrayList – Advantages of Array List over Array Wrapper classes	4 Hrs
<b>UNIT III STRINGS, CLASSES AND INTERFACES</b>		<b>18 HOURS</b>
3.1	Strings: String Array – String Methods – String Buffer Class	3 Hrs
3.2	Class and objects: Defining a class – Methods – Creating objects – Accessing class members – Constructors – Method overloading – Static members – Nesting of Methods - – this keyword – Command line input	7 Hrs
3.3	Inheritance: Defining a subclass – Deriving a sub class – Single Inheritance – Multilevel Inheritance – Hierarchical Inheritance – Overriding methods – Final variables and methods – Final classes – Final methods - Abstract methods and classes – Visibility Control: Public access, Private access, friend, protected. Interfaces: Multiple Inheritance - - Defining interface – Extending interface - Implementing Interface - Accessing interface variables	8 Hrs
<b>UNIT IV PACKAGES, APPLETS AND AWT CONTROLS</b>		<b>16 HOURS</b>
4.1	<b>Packages:</b> Java API Packages – System Packages – Naming Conventions – Creating & Accessing a Package – Adding Class to a Package – Hiding Classes	4 Hrs
4.2	<b>Applets:</b> Introduction – Applet Life cycle – Creating & Executing an Applet – Applet tags in HTML – Parameter tag – Aligning the display - Graphics Class: Drawing and filling lines – Rectangles – Polygon – Circles – Arcs – Line Graphs – Drawing Bar charts	8 Hrs
4.3	<b>AWT Components and Even Handlers:</b> Abstract window tool kit – Event Handlers – Event Listeners – AWT Controls and Event Handling: Labels – TextComponent – ActionEvent – Buttons – CheckBoxes – ItemEvent - Choice – Scrollbars – Layout Managers- Input Events – Menus	4 Hrs

UNIT–V EXCEPTION HANDLING, MULTITHREADS AND I/O STREAMS			18 HOURS
5.1	<b>Exception Handling:</b> Limitations of Error handling – Advantages of Exception Handling - Types of Errors – Basics of Exception Handling – try blocks – throwing an exception – catching an exception – finally statement	6 Hrs	
5.2	<b>Multithreading:</b> Creating Threads – Life of a Thread – Defining & Running Thread – Thread Methods – Thread Priority – Synchronization – Implementing Runnable interface – Thread Scheduling.	7 Hrs	
5.3	<b>I/O Streams:</b> File – Streams – Advantages - The stream classes – Byte streams –Character streams	5 Hrs	

### TEXT BOOKS

Sl.No.	TITLE	AUTHOR	PUBLISHER	Edition
1	Programming with Java	E. Balagurusamy	TataMc-Graw Hill, New Delhi	5 <sup>th</sup> Edition
2	Java, A Beginner's Guide	Herbert Schildt	Oracle Press	6 <sup>th</sup> Edition

<b>UNIT NO</b>	<b>TITLE</b>	<b>PAGE NOS</b>
<b>UNIT – 1</b>	<b>INTRODUCTION TO OOPS AND JAVA</b>	<b>1 TO 17</b>
<b>UNIT – 2</b>	<b>CONTROL STRUCTURES, ARRAYS, AND VECTORS</b>	<b>18 TO 41</b>
<b>UNIT – 3</b>	<b>STRINGS, CLASSES AND INTERFACES</b>	<b>42 - 73</b>
<b>UNIT – 4</b>	<b>PACKAGES, APPLETs, AWT AND EVENT HANDLERS</b>	<b>74 - 115</b>
<b>UNIT – 5</b>	<b>EXCEPTION HANDLING,MULTITHREADS AND I/O STREAMS</b>	<b>116 - 130</b>

# UNIT I INTRODUCTION TO OOPS AND JAVA

## OBJECTIVES

- To understand the basic concepts of object oriented programming features
- To know about the applications and benefits of oops
- To learn the various paradigms of programming languages
- To learn and understand about java environment.
- To learn about the creation and execution of java program.

## 1.1. Introduction to OOPS:

### 1.1.1. Paradigms of Programming Languages

Programming paradigms are a way to classify programming languages according to the style of computer programming. It provides model to the programmers to write programs. Some of the common paradigms are

1. Monolithic Programming
2. Procedural Programming
3. Structured Programming
4. Object Oriented Programming

#### Monolithic Programming

Monolithic programming is otherwise called as unstructured programming. In this paradigm the whole problem is solved as a single block. All the data are global and there is no security. To share the codes lot of goto statements are used. This is suitable only for small problem. It is difficult to correct errors.

ex : BASIC Language, Assembly Language.

#### Procedural Programming:

In procedural programming, the tasks are divided in to a number of subtasks according to their functions. These subtasks are called procedures or methods. Any procedure can be called at any point during the program execution. The program has global and local data. Data moves freely from one procedure to another. Most of the procedure share common data. Procedural programming uses Top-Down programming Approach. Here the importance is given to algorithms.

The main disadvantage of this paradigm is the difficulty in debugging and the identification of the data and its corresponding procedure.

ex : FORTRAN, Pascal.

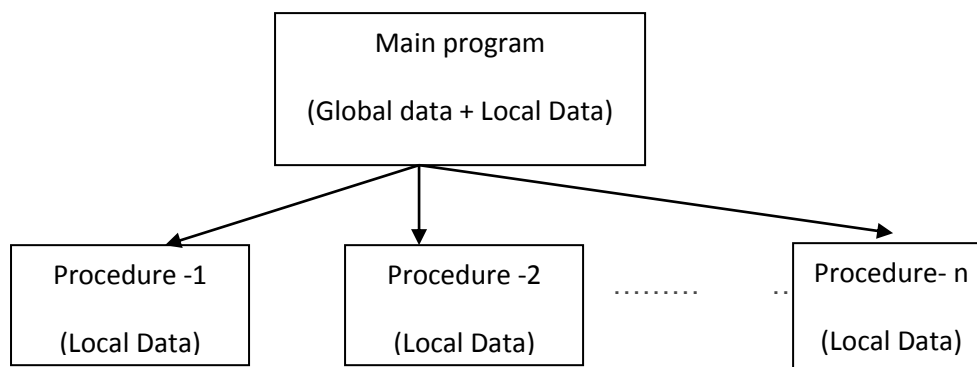
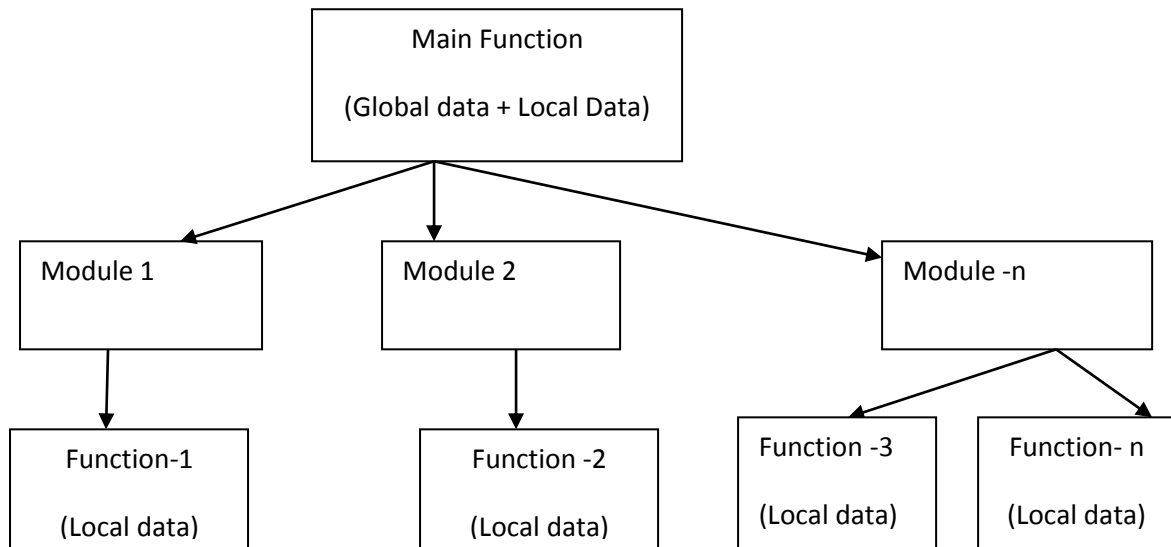


Fig :1.1. Procedural programming

### Structured Programming:

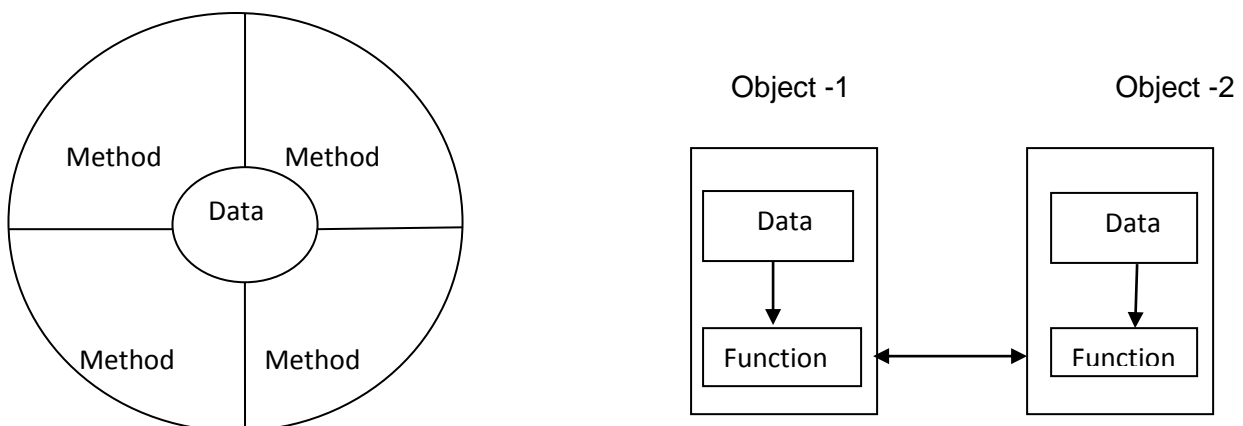
Structured programming is a powerful tool that enable the programmers to write complex programs easily. It is a subset of procedural programming. The program is divided into modules and the modules are then divided into functions. Each module works independent of one another. **C** language is a very popular structured programming language. The main disadvantage of this paradigm is when the programs grew larger this approach failed to show effective results in terms of bug free, maintenance and reusability.



**Fig :1.2. Structural programming**

### Object Oriented Programming:

The program is divided into number of small units called object. The data and function are build around these objects. The data of the objects can be accessed only by the functions associated with that object. The functions of one object can access the functions of other object. In OOP the importance is given on data rather than functions. The problems are divided into objects. Data and function are tied together. Data hiding is possible. New data and functions can be easily loaded. Object can communicate with each other using functions.



**Fig :1.3. Object = Data + Methods**



## 1.1.2. Basic concepts of Object Oriented Programming

### Objects and Classes

An **object** is an instance or result of a class. An entity that has state and behavior is known as an object. It can be physical or logical. e.g. chair, pen, table, car banking system etc. An object has three characteristics:

**state:** It represents data (value) of an object.

**behavior:** It represents the behavior (functionality) of an object such as deposit, withdraw. .

**identity:** Object identity is not visible to the external user. It is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, *color is blue* known as its *state*. It is used to write, so *writing* is its *function*.

A **class** can be defined as a template or blueprint from which the objects are created. It is a group of objects that has common properties. Class is logical entity only. Classes are user defined data types and behave like a built-in types of a programming language.

class : Student	
Name	
RollNumber	
Birthdate	
add()	
delete()	
update()	
display()	

Data ←

Methods ←

object	
Name	Anand
RollNumber	101
Birthdate	10-10-2000
add()	
delete()	
update()	
display()	

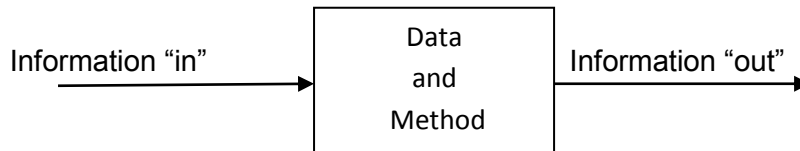
object	
Name	Prasanna
RollNumber	102
Birthdate	12-02-1990
add()	
delete()	
update()	
display()	

Fig :1.4. Representation of an object

## Data abstraction and Encapsulation

The process of hiding internal details and showing functionality is known as *abstraction*. It groups the essential details and ignore the other details. For example: phone call, we don't know the internal processing. In java, abstract class and interface is used to achieve abstraction.

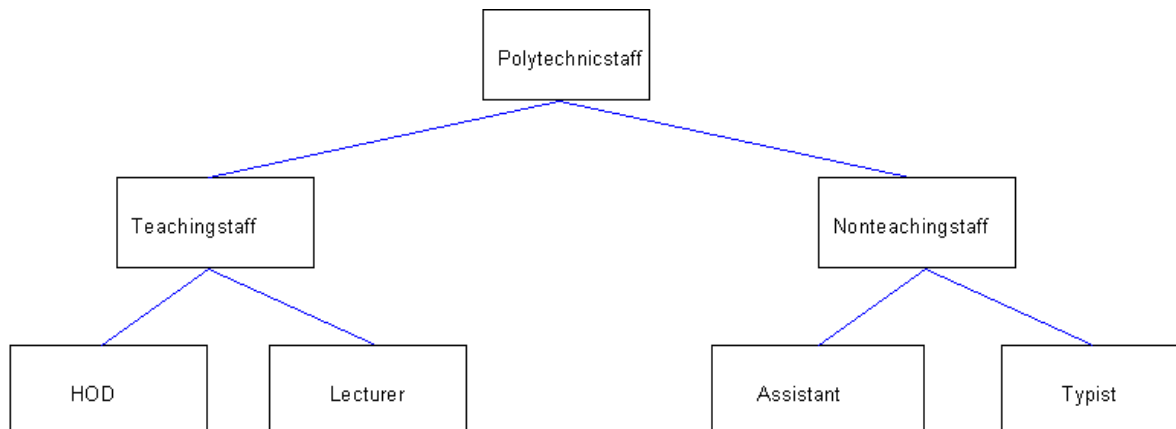
The binding or wrapping code (methods) and data together into a single unit is known as *encapsulation*. The data is not accessible to the outside world. The methods that are wrapped with the data only can access it. These methods provide the interface between the objects data and program. This insulation of the data from direct access by the program is called *data hiding*. For example: capsule, it is wrapped with different medicines. A java class is the example of encapsulation.



**Fig :1.5 Encapsulation**

## Inheritance

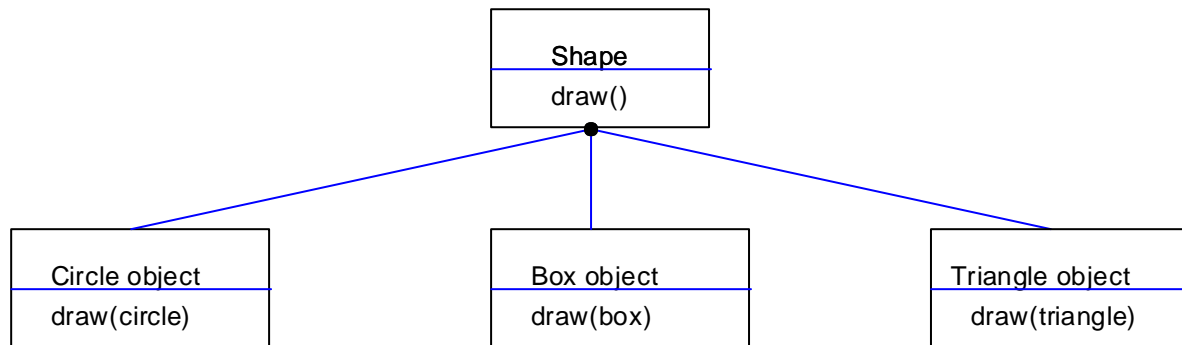
Inheritance is the process of deriving new class from the existing class. The existing class is called base class and the inherited class is called derived class. The derived class contains all the attributes and methods of the base class and also its own attributes and methods. This provides the idea of reusability.



**Fig :1.6 Inheritance**

## Polymorphism

Polymorphism means the ability to take more than one form. The same principle is applicable in object oriented programming where the objects at runtime decide what behavior will be invoked. For example, an operation may exhibit different behaviour in different instances. The behavior depends on the types of data used in the operation. For an add operation the input of two numbers will result in sum, while the input of two strings results in concatenation of the string. Polymorphism is implemented in Java using method overloading and method overriding.



**Fig :1.7 Polymorphism**

### **Dynamic binding**

Binding is defined as the linking of the procedure or function call to the corresponding program code to be executed. The binding which occurs during execution or runtime is called as dynamic binding.

### **Message communication**

An object oriented program consists of set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. It is defined as a process of sending a request to execute a function for an object. It involves specifying the name of the object, the name of the method (message) and the information to be sent. The general form is

*object\_name.message( information);*  
ex : student.add("prasanna",101,12-02-2000);

Here student is the object and add is a message sent by the object student and the information are inside the parenthesis.

### 1.1.3.Differences between Procedure Oriented Programming and Object Oriented programming

Procedure oriented Programming (POP)	Object-oriented Programming (OOP)
In procedure oriented programming the program is divided into a number of sub modules or functions or procedures.	In object oriented programming the program is divided into a number of objects.
Top-down approach is followed.	Bottom-up approach is followed.
In POP, the importance is given to algorithm and functions.	In OOP, the importance is given to data.
POP does not have any access specifier.	OOPs have access specifier such as private, public, protected etc.
Functions are independent of each other.	Each class is related in a hierarchical manner.
Maintenance is costly.	Maintenance is relatively cheaper.
Software reuse is not possible.	Helps in software reuse.
Function call is used.	Message passing is used.
Function abstraction is used.	Data abstraction is used.
No encapsulation. Data and functions are separate.	Encapsulation packages code and data altogether. Data and functionalities are put together in a single entity.
POP does not provide data hiding, so it is less secure.	OOP provides data hiding, so it is more security.
In POP overloading is not possible.	OOP provides method overloading and operator loading.
Example for POP : C, Visual Basic, Pascal etc.	Example for OOP : C++, JAVA etc

## **1.4.Benefits of OOPs**

### **a. Reusability:**

In OOP's programs functions and modules that are written by a user can be reused by other users without any modification.

### **b. Inheritance:**

It helps to eliminate redundant code and extend the use of existing classes.

### **c. Data Hiding:**

The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.

### **d. Reduced complexity of a problem:**

The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.

### **e. Easy Maintenance:**

OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.

### **f. Message Passing:**

The technique of message communication between objects makes the interface with external systems easier.

### **g. Modifiability:**

It is easy to make minor changes in the data representation or the procedures in an object oriented program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

## **1.1.5.Applications of OOPS:**

Object-oriented programming is the best paradigm to solve the complex oriented problem. The following areas make the use of OOP:

1. Image Processing and Pattern Recognition
2. Computer Aided Design and Manufacturing
3. Computer Aided Teaching
4. Intelligent Systems
5. Database Management Systems
6. Web based Applications
7. Distributed Computing and Applications
8. Component based Applications
9. Data security and management
10. Mobile Computing
11. Data Warehouse and Data Mining
12. Parallel Computing

## 1.2. JAVA:

Java is a **programming language** and a **platform**. Java is a high level, robust, secured and object-oriented programming language.

**Platform:** Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

### 1.2.1. History

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

The primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. So many different types of CPUs are used as controllers. The problem is that compilers are expensive and time-consuming to create for different types of CPU. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

### 1.2.2. Java Features

#### Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers find easy to learn Java.

#### Object-Oriented

Java is a pure object oriented language. It follows “everything is an object” paradigm. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non objects.

#### Robust

The multi platform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java.

Java is robust, because of the two main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors).

Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and de-allocation. In fact, de-allocation is completely automatic, because Java provides garbage collection for unused objects.

Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by the program.

### **Multithreaded**

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

### **Architecture-Neutral**

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

### **Interpreted and High Performance**

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

### **Distributed**

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

### **Dynamic**

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

### **Security**

As downloading a “normal” program, the downloaded program might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your

computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack. Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

### **Portability**

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed.

## **1.2.3. Java Environment**

### **Installing the Java Development Kit (JDK) and the Java Runtime Environment**

#### **Download**

There are many versions of the JDK to download, so choose the one that best suits the types of applications development. Download the latest release of the "Java SE Development Kit (JDK)". (e.g. jdk-6u18-windows-i586.exe)

#### **Installation**

Double click the executable file to begin the installation of the Java Development Kit. Accept the license agreement to continue. By choosing the default option, It will install the JDK in the system into a folder like **C:\Program Files\Java\jdk1.6.0\_18**. When the JDK has been installed, it will attempt to install the latest version of the Java Runtime Environment (JRE). Before starting on any Java development, it is essential that machine have the most current Java Runtime Environment installed.

Note that the latest version of Java is 8.

#### **Setting the PATH Environment Variable**

If default option is selected during the installation, the JDK installed itself into \Program Files\Java\jdk1.6.0\_18\bin directory.

#### **Setting the Path**

1. Right click the "My Computer" icon
2. Choose "Properties" from the popup menu
3. Click the "Advanced" tab
4. Click on the Environment Variables button
5. Under System Variables, find PATH and double click it
6. In the Edit System Variable dialog box, go to the text box labeled **Variable value:**
7. Scroll to the end of the text (hitting the <End> key will do that)
8. Insert a semicolon (;) and then the full path name of the JDK binaries directory (e.g. C:\Program Files\Java\jdk1.6.0\_18\bin)
9. Click OK to close the window(s)

Now that the JDK software is installed.



### **Popular Java Editors**

To write java programs a text editor is needed. There are even more sophisticated IDEs available in the market. some of them are

- **Notepad** – On Windows machine, any simple text editor like Notepad ,textpad can be used.
- **Netbeans** – A Java IDE that is open-source and free which can be downloaded from <https://www.netbeans.org/index.html>.
- **Eclipse** – A Java IDE developed by the eclipse open-source community and can be downloaded from <https://www.eclipse.org/>.

### **1.2.4.JDK (Java Development kit)**

JDK contains tools that are required to build and execute Java applications and applets. JDK can be freely downloaded from the sun's website.

#### **javac - Java Compiler**

The Java source code will be compiled by this compiler, which checks for the syntax of the program and will list out errors if any, or will generate the byte code. The Java compiler generate the .class file (byte code).

#### **java – Java interpreter**

Java interpreter reads the .class file of the application programs that contains the byte code and executes the code in a local machine.

#### **appletviewer**

The applet viewer runs the Java applets

#### **javadoc -Java Documentation**

Document generator is used to create documents for large source codes.

#### **javah - C header**

It produces header files that are used for writing native methods.

#### **javap - Disassembler**

It is used to convert byte code into program description.

#### **jdb- Debugger**

It is used to find errors in the program.

### **1.2.5.API (Application Programming Interface)**

Java API is a set of classes and interfaces that comes with the JDK. Java API is actually a huge collection of library routines that performs basic programming tasks such as looping, displaying GUI form etc. In the Java API, classes and interfaces are packaged in packages. All these classes are written in Java programming language and runs on the JVM. Java classes are platform independent but JVM is not platform independent. So there are different downloads for each OS. The commonly used packages are

#### **a.lang package**

A collection of classes and methods which includes the basic features of java. This is a default package.

#### **b. util package**

A collection of classes and methods for providing date and time.

#### **c.applet package**

A collection of classes and methods for creating and running applets.

#### **d.abstract window toolkit package**

A collection of classes and methods for implementing graphical user interface.

#### **e.input/output package.**

A collection of classes and methods for input output operations.

## 1.3.Introduction to Java :

### 1.3.1. Types of java program

There are two types of Java programs. They are as follows :

1. Application Programs
2. Applet Programs

#### **Application Programs**

Application programs are stand-alone programs that are written to carry out certain tasks on local computer such as solving equations, reading and writing files etc. The application programs are allowed to access the local file system and the resources. The application programs can be executed using two steps

1. Compile source code to generate Byte code using Javac compiler.
2. Execute the byte code program using Java interpreter.

#### **Applet programs**

Applets are small Java programs developed for Internet applications. An applet located in distant computer can be downloaded via Internet and executed on a local computer using Java enabled browser. The applets are restricted to access the local file system and the resources. The Java applets can also be executed in the command line using appletviewer, a JDK tool.

### 1.3.2.Creating and Executing a Java program

#### 1. Creating a program :

In Java, a source file is officially called a *compilation unit*. It is a text file that contains one or more class definitions. Using any text editor such as notepad in windows or edit in DOS etc , a java source file can be created. The created java file should be saved with extension as java.

The general form is **filename.java**

If the source file has more than one class, then the filename should be the class name which is having the main method in it.

The general form of a java source file is

```
class classname
{
    // Your program begins with a call to main().
    public static void main(args[ ])
    {
    }
}
```

Here class name is user defined and class is a keyword.

#### **example**

```
class First
{

    public static void main(args[ ])
    {
        System.out.println("Welcome to Java World");
    }

}
```

Save the file as **First.java**

## 2.Compiling the program

javac compiler is used to compile the java program. The java source program is compiled into class files. Java compiler creates a class file and the class files contains the byte codes of the program.

The general form for compiling is

```
javac filename.java
```

example :

In the command prompt type

```
C:\> javac First.java
```

After compilation **First.class** file is generated.

## 3.Executing the program

java interpreter is used to execute the compiled program.The general form for executing a java file is

```
java classname
```

example :

In the command prompt type

```
C:\> java First
```

When the program is run, the following output is displayed for the above program First.java:

```
Welcome to Java world
```

### 1.3.3.Java Tokens:

The Smallest individual elements in a program are called tokens. Java language includes five types of tokens. More than one token can appear in a same line separated by spaces. They are as follows

- 1.Keywords
- 2.Identifiers
- 3.Literals
- 4.Operators
- 5.Separators

example :

```
class First
{
    int x,y,z;
}
keyword – class
identifier – x y z First
separators - , ; { }
```

### Keywords

keywords are the words which belongs to the java language. They have standard predefined meaning. They should be used only for the intended purpose. They should be written in small case letters. Here is a list of keywords in the Java programming language. They cannot be used as identifiers in the programs. The keywords const and goto are

reserved, even though they are not currently used. true, false, and null might seem like keywords, but they are actually literals;

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

### Character set

A character set is a set of textual and graphic symbols, each of which is mapped to a set of nonnegative integers. Character set defines the characters that are used in a programming language. Java uses unicode character set. Unicode is a 16-bit character set designed to cover all the world's major living languages, in addition to scientific symbols and dead languages that are the subject of scholarly interest. The first 256 values are the same as the ASCII character set.

### Identifiers

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number. Java is case-sensitive, so **VALUE** is a different identifier than **Value**.

Some examples of valid identifiers are

```
AvgTemp
count
a4
$test
this_is_ok
```

Invalid identifier names include these:

```
2count      // Cannot start with numeral
high-temp   // Cannot have hyphen symbol
Not/ok      // operators are not allowed as identifier
My Class    // spaces are not allowed
```

### Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

```
Integer literals: 10 15 2
Floating-point literals: 2.3 1.1
Character literals: ' ' 'J' 'j'
Boolean literals: true false
String literals: "Java" "100" "x"
```

### Separator

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. It is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parenthesis	Used to contains list of parameters
{ }	Braces	Used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
;	Semicolon	Terminating statements
.	Period	Used to separate package names from sub packages and classes. Also used to separate a variable or method from a reference variable.

### 1.3.4.Java Virtual Machine (JVM)

A **Java virtual machine (JVM)** is an abstract computing **machine** that enables a computer to run a **Java** program. There are three notions of the **JVM** specification, implementation and instance. The specification is a document that formally describes what is required of a JVM implementation. Having a single specification ensures all implementations are interoperable. A JVM implementation is a computer program that meets the requirements of the JVM specification. An instance of a JVM is an implementation running in a process that executes a computer program compiled into Java bytecode.

Java compiler compiles the source code into bytecode which is a machine independent code. These intermediate codes are called java virtual machine (JVM). These intermediate codes are converted into machine dependent code with the help of interpreter and can be run by the computer.

### 1.3.5.Command Line Arguments

A Java application can accept any number of arguments from the command line. The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run. When an application is launched, the runtime system passes the command-line arguments to the application's main method via an array of Strings.

For example, suppose a Java application called student requires name list, the user would enter the names in the command line

```
public class student {  
    public static void main (String[] args) {
```

```

        for (String s: args) {
            System.out.println(s);    }    }
    }

```

The following example shows how a user might run **student**. User input is in italics. **student** is a class name

```
java student Anand Prasanna Raja
```

```
Anand
```

```
Prasanna
```

```
Raja
```

The application displays each word — *Anand*, *Prasanna*, *Raja* — on a line by itself. This is because the space character separates command-line arguments.

**args.length** will provide the total number of arguments passed.

### 1.3.6. Comments in Java program

Comments are non executable statements included in the program for understanding purpose. In java language there are three kinds of comments. They are as follows

1. Single line comment
2. Multiline comment
3. Documentation comment

#### 1. Single line comment

Single line comment starts with **//**.

```
// text
```

The compiler ignores everything from **//** to the end of the line.

#### 2. Multiline comment

Multiline comment starts with **/\*** and ends with **\*/**.

```
/* text */
```

The compiler ignores everything from **/\*** to **\*/**.

#### 3. Documentation comment

Documentation comment starts with **/\*\*** and ends with **\*/**.

```
/** documentation */
```

The compiler ignores everything from **/\*\*** to **\*/**. The JDK javadoc tool uses doc comments when preparing automatically generated documentation.

### example

The bold characters in the following listing are comments.

```

/*
    The HelloWorldApp class implements an application that
    simply displays "Hello World!" to the standard output.
*/
class HelloWorldApp
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!"); //Display the string.
    }
}

```

## **Review Questions**

### **Short answer Questions**

1. What are the paradigms of programming languages?
2. Define class.
3. Define object.
4. What is mean by data encapsulation?
5. What is mean by data abstraction?
6. Define inheritance.
7. Define polymorphism.
8. What is mean by dynamic binding?
9. How the objects are communicated?
10. Expand JDK and API.
11. What are the types of java program?
12. What are java tokens?
13. Define literal.
14. What are the character set used in java?
15. Define comments.
16. What are the types of comments in a java program?
17. What is mean by identifier?
18. What is keyword in java?
19. What are the available separators in java?
20. What are command line arguments?

### **Long answer questions**

1. Explain in detail about the various paradigms used in programming.
2. What are the benefits of OOPS?
3. What are the application of OOPS?
4. Explain with an example about object and classes.
5. Explain about the history of java.
6. What are the features of java ? Explain them.
7. Explain about the java environment.
8. Explain about the JDK and API.
9. How a java program is created and executed? Explain in detail.
10. Explain in detail about the JVM.
11. Explain with example about the command line arguments.
12. Describe about the various comments used in a java program.

## UNIT II CONTROL STRUCTURES, ARRAYS, AND VECTORS

### OBJECTIVES

To understand the concepts of variables, constants  
To learn the various operators and evaluation of expressions  
To learn the control structures and looping statements  
To learn and understand about arrays and vectors.

### 2.1. Elements:

#### 2.1.1.Constants :

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100 98.6 'X' "This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

#### 2.1.2.Variables :

Variable is name of reserved area allocated in memory. Its value changes during execution.



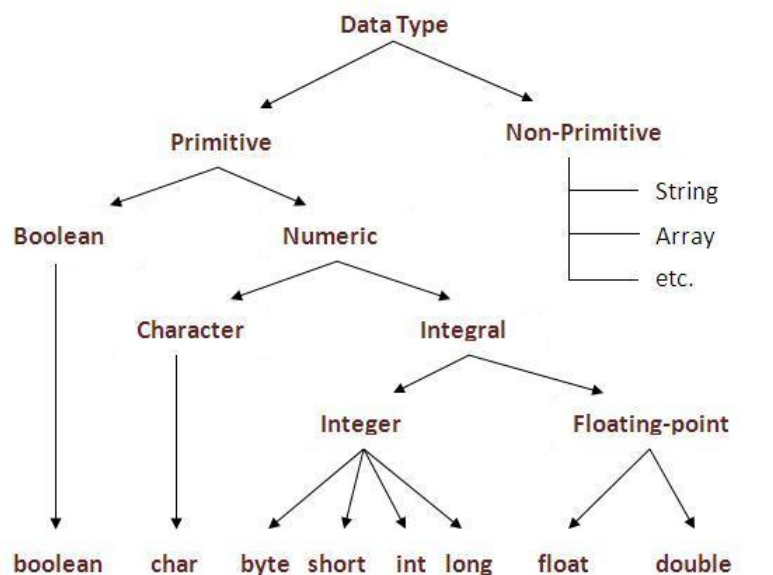
`int x=50; //Here x is variable`

#### 2.1.3.Datatypes :

In java, the data types are classified into

- primitive data types
- non-primitive data types





Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types. These can be put in four groups:

- **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.
- **Characters** This group includes **char**, which are symbols in a character set, like letters and numbers.
- **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

### Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values.

#### byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

#### short

**short** is a signed 16-bit type. It has a range from –32,768 to 32,767. Here are some examples of **short** variable declarations:

```
short s; short t;
```

#### int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647.

#### long

**long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large

enough to hold the desired value. The range of a **long** is quite large.

### Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively.

#### **float**

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Here are some example **float** variable declarations:

```
float hightemp, lowtemp;
```

#### **double**

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.

#### **Characters**

In Java, the data type used to store characters is **char**. For this purpose, it requires 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536.

#### **Booleans**

Java has a primitive type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, as in the case of **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

### **2.1.4. Scope of Variables**

Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope*. A scope determines what objects are visible to other parts of the program. It also determines the lifetime of those objects. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when a variable is declared within a scope, that variable is a **local variable** and protecting it from unauthorized access and/or modification.

Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class Scope {
    public static void main(String args[]) {
        int x;           // known to all code within main
        x = 10;
        if(x == 10) {    // start new scope
            int y = 20;  // known only to this block
                        // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100;      // Error! y not known here
                        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

As the comments indicate, the variable **x** is declared at the start of **main( )**'s scope and is accessible to all subsequent code within **main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is only visible to other code within its block. This is why

outside of its block, the line **y = 100;** is commented out. If the leading comment symbol is removed, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

### 2.1.5.Type casting

It is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types by using *cast*, which performs an explicit conversion between incompatible types.

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other. Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

To create a conversion between two incompatible types, use a cast. **A cast is simply an explicit type conversion. It has this general form:**

**(target-type) value**

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

### 2.1.6.Operators:

#### Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
−	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement

The operands of the arithmetic operators must be of a numeric type.

#### Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

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

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, **==**, and the inequality test, **!=**. As stated, the result produced by a relational operator is a **boolean** value. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of **a < b** (which is **false**) is stored in **c**.

### The Assignment Operator

The *assignment operator* is the single equal sign, **=**. The assignment operator works in Java much as it does in any other computer language. It has this general form:

*variable = expression;*

Here, the type of *variable* must be compatible with the type of *expression*. The assignment operator allows to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a "chain of assignment" is an easy way to set a group of variables to a common value.

### Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
	Logical OR
^	Logical XOR (exclusive OR)
!	Logical unary NOT

The logical Boolean operators, **&**, **|**, and **^**, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state : **!true == false** and **!false == true**. The following table shows the effect of each logical operation:

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

## The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left

Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value. Specifically, it is useful to know how Java stores integer values and how it represents negative numbers. All of the integer types are represented by binary numbers of varying bit widths. For example, the **byte** value for 42 in binary is 00101010, where each position represents a power of two, starting with 20 at the rightmost bit. The next bit position to the left would be 21, or 2, continuing toward the left with 22, or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of  $2^1 + 2^3 + 2^5$ , which is  $2 + 8 + 32$ . All of the integer types (except **char**) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as *two's complement*, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example, -42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or -42. To decode a negative number, first invert all of the bits, then add 1. For example, -42, or 11010110 inverted, yields 00101001, or 41, so when 1 is added, the result will be 42.

## The Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome of each operation.

A	B	A   B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

### The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, **~**, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

```
00101010
becomes
11010101
```

after the NOT operator is applied.

### The Bitwise AND

The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
00101010 42
& 00001111 15
00001010 10
```

### The Bitwise OR

The OR operator, `|`, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
00101010    42
| 00001111    15
00101111    47
```

### The Bitwise XOR

The XOR operator, `^`, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the `^`. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```
00101010    42
^ 00001111    15
00100101    37
```

### The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the `?`. The `?` has this general form:

*expression1 ? expression2 : expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the `?` operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**.

Here is an example of the way that the `?` is employed:

```
ratio = denom == 0 ? 0 : num / denom;
```

When Java evaluates this assignment expression, it first looks at the expression to the *left* of the question mark. If **denom** equals zero, then the expression *between* the question mark and the colon is evaluated and used as the value of the entire `?` expression. If **denom** does not equal zero, then the expression *after* the colon is evaluated and used for the value of the entire `?` expression. The result produced by the `?` operator is then assigned to **ratio**.

### 2.1.7.SPECIAL OPERATOR

#### 1. instanceof Operator

This operator is used to know if an object is an instance of a particular class or not.

This operator returns "true" if an object given at the left hand side is an instance of the class given at right hand side. Otherwise, it returns "false".

##### Example

```
circle instanceof Shape
```

The above statement checks if the object "circle" is an instance of the class "Shape".

If yes, it returns "true", else returns "false"

#### 2. Dot operator

This operator is used to access the variables and methods of a class.

##### Example 1

```
student.mark
```

Here we are accessing the variable "mark" of the "student" object

##### Example 2

```
student.getMarks()
```

Here we are accessing the method "getMarks" of the "student" object.

This operator also can be used to access classes, packages etc.

## 2.1.8.Expressions- Evaluation of Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

```
int a = 0;  
int b = 100;  
int c;  
c= a+ b;
```

```
System.out.println("c: " + c);
```

```
if (value1 == value2)  
    System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `c = a + b` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, `c` is an `int`. As from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

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

Table : The precedence of java operators.

Parentheses are used to alter the precedence of an operation. The square brackets provide array indexing. The dot operator is used to dereference objects. *Parentheses* raise the precedence of the operations that are inside them.

The Java programming language allows to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other. Here's an example of a compound expression:

`1 * 2 * 3`

In this particular example, the order in which the expression is evaluated is unimportant because the result of multiplication is independent of order; the outcome is always the same, no matter in which order you apply the multiplications. However, this is not

true of all expressions. For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

```
x + y / 100 // ambiguous
```

To specify exactly how an expression will be evaluated using balanced parenthesis: For example, to make the previous expression unambiguous, you could write the following:

```
(x + y) / 100 // unambiguous, recommended
```

If the order is not mentioned explicitly for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

```
x+y/100
```

```
x + (y / 100) // unambiguous, recommended
```

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first.

## 2.2 Decision making and Branching:

A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allows program to execute in a nonlinear fashion.

### 2.2.1.The Simple if Statement

The Java **if** statement works much like the IF statement in any other language. Its syntax is shown here:

```
if(condition) statement;
```

Here, *condition* is a Boolean expression. If *condition* is true, then the statement is executed.If *condition* is false, then the statement is bypassed. Here is an example:

```
if(num < 100) System.out.println("num is less than 100");
```

In this case, if **num** contains a value that is less than 100, the conditional expression is true, and **println( )** will execute. If **num** contains a value greater than or equal to 100, then the **println( )** method is bypassed.



### 2.2.2.if else statement

The **if else** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if else** statement:

```
if (condition) statement1;
else statement2;
```

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional. The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

### 2.2.3.Nested if

A *nested if* is an **if** statement that is the target of another **if** or **else**. Nested **ifs** are very common in programming. When **ifs** are nested, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```
if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d; // this if is
    else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

### 2.2.4.The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **ifs** is the *if-else-if ladder*. It looks like this:

```
if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
...
else
    statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed.

The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

Here is a program that uses an **if-else-if** ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
        int month = 4; // April
        String season;
        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";
        System.out.println("April is in the " + season + ".");
    }
}
```

Here is the output produced by the program:

April is in the Spring.

### 2.2.5.switch

The **switch** statement is Java's multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    ...
    case valueN:
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the

expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken. The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of “jumping out” of the **switch**. Here is a simple example that uses a **switch** statement:

```
// A simple example of the switch.
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                case 3:
                    System.out.println("i is three.");
                    break;
                default:
                    System.out.println("i is greater than 3.");
            }
    }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

## Looping Statements

Java's iteration statements or looping statements are **for**, **while**, and **do-while**. These statements create *loops*. A loop repeatedly executes the same set of instructions until a termination condition is met. Java has a loop to fit any programming need.

### 2.2.6.While

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {  
    // body of loop  
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Here is a **while** loop that counts down from 10, printing exactly ten lines of "tick":

```
// Demonstration of while loop.  
class While {  
    public static void main(String args[]) {  
        int n = 10;  
        while(n > 0) {  
            System.out.println("tick " + n);  
            n--;  
        }  
    }  
}
```

When you run this program, it will "tick" ten times:

```
tick 10  
tick 9  
tick 8  
tick 7  
tick 6  
tick 5  
tick 4  
tick 3  
tick 2  
tick 1
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println( )** is never executed:

```
int a = 10, b = 20;  
while(a > b)  
    System.out.println("This will not be displayed");
```

### 2.2.7.do-while

If the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {  
    // body of loop
```

```
    } while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, *condition* must be a Boolean expression.

Here is an example that demonstrates the **do-while** loop.

It generates the same output as before.

```
// Demonstration of do-while loop.
class DoWhile {
    public static void main(String args[]) {
        int n = 10;
        do {
            System.out.println("tick " + n);
            n--;
        } while(n > 0);
    }
}
```

### 2.2.8.for

It is a powerful and versatile construct. Here is the general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {
    // body
}
```

If only one statement is being repeated, there is no need for the curly braces. The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Here is an example uses a **for** loop:

```
// Demonstration of for loop.
class ForTick {
    public static void main(String args[]) {
        int n;
        for(n=10; n>0; n--)
            System.out.println("tick " + n);
    }
}
```

### 2.2.9.break

In Java, the **break** statement has three uses. First, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto. By using **break**, immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop is forced. When a

**break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

This program generates the following output:

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

Loop complete.

### 2.2.10. Labelled loop

The statement **continue** may specify a **label** to describe which enclosing loop to continue. Here is an example program that uses **continue** to print a triangular multiplication table for 0 through 9. The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**.

```
// Using continue with a label.
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
            System.out.println();
        }
    }
}
```

Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
```

```
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

### 2.2.11.continue

Sometimes it is useful to force an early iteration of a loop. That is, it might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. Here is an example program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

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

Good uses of **continue** are rare. One reason is that Java provides a rich set of loop statements which fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

## 2.3 ARRAYS

### 2.3.1 Arrays

An array is a collection of variables of the same type, referred by a common name. The number of values that can be stored in an array is known as size of the array. The size of the array must be integer and cannot be changed during run time. The individual value of an array is called element. A specific element in an array is accessed by writing a number is known as index or subscript. An index always start with zero.

#### 2.3.1.1 One Dimensional Array

An array with only one subscript is known as one dimensional array.

**eg:** If we want to store a set of marks for a student, then we use one dimensional array.

```
Int mark[ ] = { 70,68,72,60,75,74};
```

The elements of an array are stored in contiguous locations of memory.

70
68
72
60
75
74

#### 2.3.1.2 Creating an Array

Arrays must be declared and created in the computer memory before they are used.

Creation of an array involves three steps. They are

- (i). Declaring an array
- (ii). Allocate memory space
- (iii). Store the values

##### (i) Declaring the array

It may be declared in two ways.

```
datatype arrayname[ ]; or      datatype [ ] arrayname;  
where
```

datatype – valid datatype

arrayname - name of the array

**eg:** int x[ ]; or int [ ] x;

##### (ii) Allocate memory space

The memory space of an array is allocated using new operator.

```
arrayname = new datatype[size];
```

where

datatype – valid datatype

arrayname - name of the array

size - specifies the total number of elements to be stored

new – operator to allocate memory space

**eg:** x = new int [10];

##### (iii) Store the values

After creating an array, the values should be stored in it using the following syntax

```
arrayname [index] = value;
```

where

arrayname - name of the array

index- specifies the position of individual element

value- constant

**eg:** x[1]=40;

NOTE : we also create an array using the following syntax.

```
datatype arrayname[ ]= new datatype[size];
```

**eg:** int y[ ]= new int[5];



### 2.3.1.3 Array processing

We know that array is a group of data of same data type, so to use the array elements we use any one of the looping statements. Normally for loop is used.

**Example:** To find the average of array elements

```
public class Example1
{
    public static void main(String args[ ])
    {
        double sum=0,avg;
        int x[]={22,32,42,52,62};
        for( int i=0; i<5; i++)
            sum= sum+x[i];
        avg = sum/5;
        System.out.println("Average of array elements are:"+avg);
    }
}
```

**Example:** To find the largest array element

```
public class Example2
{
    public static void main(String args[ ])
    {
        int max;
        int x[]={122,12,40,5,68};
        max = x[0];
        for( int i=1; i<5; i++)
            if (x[i]> max) max= x[i];
        System.out.println("Largest array element is :"+max);
    }
}
```

### 2.3.1.4 Multi Dimensional Array

In Java, a multidimensional array is an array of arrays. An array with more than one subscripts is known as multi dimensional array.

datatype arrayname[ ][...] = new datatype[size 1] [size 2] ....[size n];

**eg:** int a[ ][ ][ ] = new int [3][3][3];

**Example:** To read and display the elements of matrix 2 x 2

```
import java.io.*;
public class Example3
{
    public static void main(String args[ ]) throws IOException
    {
        int x[][] = new int[2][2];
        int i,j;
        DataInputStream d1= new DataInputStream(System.in);
        System.out.println("Enter array elements one by one");
        for( i=0; i<2; i++)
            for( j=0; j<2; j++)
                x[i][j]= Integer.parseInt(d1.readLine());
        System.out.println("Matrix elements ");
        for( i=0; i<2; i++)
            for( j=0; j<2; j++)
                System.out.print("\t"+x[i][j]);    }}
}
```

### 2.3.2 Vectors

#### Vector

Vector is a class which is used to create dynamic array. Dynamic array means size can be changed at run time. It also stores different types of objects. Vector class is defined in **java.util** package.

#### Creation of Vector

A Vector can be created in three ways.

Description	Syntax	Example::
To create an empty vector with the default initial capacity of 10.	Vector vectorname = new Vector( );	Vector v1= new Vector ( );  It creates vector v1 of capacity 10.
To create a vector with the given capacity	Vector vectorname = new Vector(capacity);	Vector v2= new Vector (5 );  It creates vector v2 of given capacity 5.
To create a vector with the given capacity & increment.	Vector vectorname = new Vector(capacity,n);	Vector v3= new Vector (5,2 );  It creates a vector v3 of given capacity 5 and Increment is 2.

#### Methods

All the vector methods are called using the syntax  
Vectorobject.methodname();

Sl. No	Method name	Use	Example::
1.	int capacity()	This method returns the maximum number of objects to be stored in the vector	Vector v2= new Vector (5 ); v2.capacity();
2.	int size()	It returns the number of objects present in the vector	Vector v2= new Vector (5 ); v2.size();
3.	void addElement(Object )	It inserts the object at the end of the vector.	Vector v2= new Vector (5 ); v2.addElement("Fruits");
4.	void insertElement(Object,n )	It inserts the object at the nth position of the vector.	Vector v2= new Vector (5 ); v2.insertElement("Mango",4);
5.	Object elementAt(int n)	It returns the element present at the nth position of the vector	Vector v2= new Vector (5 ); v2.elementAt(2);
6.	boolean removeElement(Object )	It removes the specified object from the vector.	Vector v2= new Vector (5 ); v2.removeElement("Fruits");
7.	boolean removeElementAt (n)	It removes the object at the nth position of the vector.	Vector v2= new Vector (5 ); v2.removeElementAt(3);
8.	boolean removeAllElements( )	It removes all the	Vector v2= new Vector (5 );

		elements of the vector	v2.removeAllElements( );
9.	copyInto(arrayname)	It copies all the objects of vector into array	Vector v2= new Vector (5 ); v2.copyInto(x);

#### Example:

```
import java.util.*;
public class Example4
{
    public static void main(String args[ ])
    {
        Vector v1 = new Vector(2);
        System.out.println("Initial Size is: "+v1.size());
        System.out.println("Initial capacity is: "+v1.capacity());
        v1.addElement("C");
        v1.addElement("C++");
        v1.addElement("Java");
        System.out.println("Size after adding elements: "+v1.size());
        System.out.println(" capacity after adding elements: "+v1.capacity());
        v1.insertElementAt("VB",2);
        for(int i=0; i<v1.size();i++)
            System.out.println("\t Elements are:" +v1.elementAt(i));
        v1.removeElementAt(1);
        System.out.println("Size after removing element: "+v1.size());
        for(int i=0; i<v1.size();i++)
            System.out.println("\t Elements are:" +v1.elementAt(i));
    }
}
```

#### Difference between Array and Vector

Array	Vector
Size of the array cannot be changed	Size of the vector can be changed
Values of same data types can be stored	Objects of different data types can be stored

### 2.3.3 ArrayList

ArrayList class is used to create dynamic array. Dynamic array means, the size of the array can be changed at the run time.

Array lists are created with an initial size. When this size is exceeded, it is automatically enlarged. When objects are removed, the array may be shrunk. It is defined in **java.util** package.

#### Creation

An ArrayList can be created in two ways.

Description	Syntax	Example::
To create an empty array list with the default initial capacity of 10.	ArrayList name = new ArrayList ( );	ArrayList a1= new ArrayList ( );  It creates ArrayList a1 of capacity 10.

#### Methods

All the ArrayList methods are called using the syntax  
ArrayListname.methodname();

Sl. No	Method name	Use	Example::
1.	int size()	This method returns the number of elements in this list.	ArrayList a1= new ArrayList ( ); a1.size();
2.	boolean add(Object)	It inserts the object at the end of the list.	ArrayList a1= new ArrayList ( ); a1.add("oops");
3.	void add(int n, Object)	It inserts the object at the nth position of the list	ArrayList a1= new ArrayList ( ); a1.add(3,"oops");
4.	Object get(int n)	It returns the element at the nth position in this list.	ArrayList a1= new ArrayList ( ); a1.get(2);
5.	Object set(int n, Object)	It replaces the element present at the nth position with the given object.	ArrayList a1= new ArrayList ( ); a1.set(3,"java");
6.	Object remove(int n)	It removes the element at the nth position of the list.	ArrayList a1= new ArrayList ( ); a1.remove(2);
7.	void clear()	It removes all of the elements from this list.	ArrayList a1= new ArrayList ( ); a1.clear();

**Example:**

```
import java.util.*;
public class Example5
{
    public static void main(String args[] )
    {
        ArrayList a1 = new ArrayList();
        System.out.println("Initial Size is: "+a1.size());
        a1.add("C");
        a1.add("C++");
        a1.add("Java");
        System.out.println("Size after adding elements: "+a1.size());
        a1.add(2,"VB");
        System.out.println("After adding element: ");
        for(int i=0; i<a1.size();i++)
            System.out.println("\t Elements are:" +a1.get(i));
        a1.remove(0);
        a1.set(1,"c#");
        System.out.println("After removing & changing element: ");
        for(int i=0; i<a1.size();i++)
            System.out.println("\t Elements are:" +a1.get(i));
        a1.clear();
        System.out.println("Size after removing element: "+a1.size());
    }
}
```

### Advantages of ArrayList over Array

1. Its size can be changed at the run time.
2. It stores different type of objects.
3. It has many methods to manipulate the stored objects.

### 2.3.4 Wrapper class

Wrapper class in java provides the mechanism to convert primitive data type into object and object into primitive data type. It is defined in **java.lang** package.

PRIMITIVE DATA TYPE	WRAPPER CLASS
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Using wrapper classes, we can perform the following conversions:

1. Converting primitive data type to object
2. Converting object to primitive data type
3. Converting primitive data type to string object
4. Converting string object to primitive data type
5. Converting string object to number object

#### 1. Converting primitive data type to object

To Convert primitive data type to object, **constructor** is used.

classname objectname = new classname(value);

- eg:** 1. Convert int to integer object ----- Integer k1= new Integer (25);  
2. Convert double to double object ----- Double m1= new Double(125.98);

#### 2. Converting object to primitive data type

To convert object to primitive data type, **typeValue()** is used. Here type specifies the primitive datatype.

datatype variabelname = objectname. typeValue();

- eg:** 1. Convert Integer to int object ----- int p1= k1. intValue();  
2. Convert float object to float ----- float a1= m1.floatvalue();

### 3. Converting primitive data type to string object

To Convert primitive data type to stringobject, **toString()** is used.

String objectname = classname.toString(value);

- eg:** 1. Convert int to string object ----- String s1= Integer.toString(57);  
2. Convert double to string object ----- String s3= Double.toString(57.3421);

### 4. Converting string object to primitive data type

To convert object to primitive data type, **parseType()** is used. Here type specifies the primitive datatype.

datatype variablename = classname.parseType(stringobject);

- eg:** 1. Convert string object to int ----- int p1= Integer.parseInt(s1);  
2. Convert string object to float ----- float a1= Float.parseFloat(s3);

### 5. Converting string object to number object

To Convert string object to number object, **valueOf()** method is used.

classname objectname = classname.valueOf(stringobject);

- eg:** 1. Convert string object to Integer object ---- Integer k1= Integer.valueOf(s1);  
2. Convert string object to Double object ---- Double m1= Double.valueOf(s3);

**REVIEW QUESTIONS**  
**PART - A**

1. What is a variable?
2. Define constant.
3. What is mean by type casting?
4. What is the use of break statement?
5. What is the use of continue statement?
6. Define array.
7. What is one dimensional array?.
8. What is array processing?.
9. Define vector.
10. What is the use of addElement()?.
11. List the differences between array and vector.
12. What is ArrayList?
13. List the advantages of arraylist over array.
14. What is the use of clear()?.
15. What is wrapper class?.
16. List any 4 wrapper classes.

**PART - B**

1. Write notes on scopes of variables.
2. What are labeled loops?
3. Write the difference between while and do while statement?
4. Define multidimensional array. Explain
5. Write a note on creating a vector.
6. Which methods are used to delete elements from vector? Explain.
7. Write a note on creating an arraylist.
8. Name the methods used to add elements to an arraylist and explain.

**PART - C**

1. Explain the various control statements in java.
2. Explain the various operators available in java.
3. Explain the switch statement with example.
4. What are the looping statements available in java ? Explain.
5. Explain the creation of one dimensional array.
6. Write a program to find the smallest elements in an array.
7. Write a program to find the average of N elements in an array.
8. Explain in detail about methods of Vector class.
9. Explain methods of ArrayList class.
10. Explain about wrapperclass.

## UNIT III STRINGS, CLASSES AND INTERFACES

### OBJECTIVES

To Understand the concepts of strings  
To know to work with string methods  
To learn the creation of classes and objects  
To know about the interfaces.

### 3.1 Strings:

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
char[] ch={'j','a','v','a'};
```

String s=new String(ch); is same as:

```
String s="java";
```

**String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming. The first thing to understand about strings is that every string is actually an object of type **String**. Even string constants are actually **String** objects.

For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** constant.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

#### 3.1.1.String Array

A Java String Array is an object that holds a fixed number of String values.

##### String Array Declaration

Square brackets is used to declare a String array. There are two ways of using it. The first one is to put square brackets after the String reserved word. For example:

```
String[] thisIsAStringArray;
```

Another way of declaring a String Array is to put the square brackets after the name of the variable. For example:

```
String thisIsAStringArray[];
```

Both statements will declare the variable "thisIsAStringArray" to be a String Array. Note that this is just a declaration, the variable "thisIsAStringArray" will have the value null. And since there is only one square brackets, this means that the variable is only a one-dimensional String Array.

##### String Array Declaration With Initial Size

Arrays are usually used when how many objects are needed. Hence, arrays are usually declared with an initial size. Here is an example:

```
String[] thisIsAStringArray = new String[5];
```



This will declare a String Array with 5 elements. Each element can be accessed using index that starts with 0. The first element is "thisIsAStringArray[0]", the second element is "thisIsAStringArray[1]", and so on. Here is an example of declaring a String Array with size 5 and assigning values to each element:

```
String[] thisIsAStringArray = new String[5];
thisIsAStringArray[0] = "AAA";
thisIsAStringArray[1] = "BBB";
thisIsAStringArray[2] = "CCC";
thisIsAStringArray[3] = "DDD";
thisIsAStringArray[4] = "EEE";
```

Note, since there are only 5 elements and index started from 0, the last index will be 4. Or we could use the formula ( array length - 1 )

#### **String Array Initialization on Declaration**

Initialization can also be done at the same time as the declaration. Here is an example:

```
String[] thisIsAStringArray = {"AAA", "BBB", "CCC", "DDD", "EEE"};
```

This will create a String Array of length 5. Element at index 0 will have the value "AAA", element at index 1 will have the value "BBB", and so on. Hence, when we run this code:

```
String[] thisIsAStringArray = {"AAA", "BBB", "CCC", "DDD", "EEE"};
System.out.println( thisIsAStringArray[0] );
System.out.println( thisIsAStringArray[1] );
System.out.println( thisIsAStringArray[2] );
System.out.println( thisIsAStringArray[3] );
System.out.println( thisIsAStringArray[4] );
```

It will produce this output:

```
AAA
BBB
CCC
DDD
EEE
```

Another syntax to declare and initialize a String array together is by using the new operator. Here is an example:

```
String[] thisIsAStringArray = new String[]{"AAA", "BBB", "CCC", "DDD", "EEE"};
```

The behaviour is the same as the example above:

```
"String[] thisIsAStringArray = {"AAA", "BBB", "CCC", "DDD", "EEE"};"
```

#### **String Array Initialization After Declaration**

The code for initializing an array can be separated from the declaration code. This is useful when values are not known during declaration. Here is an example:

```
String[] thisIsAStringArray;
if (fruits) {
    thisIsAStringArray = new String[] {"Apple", "Banana", "Orange"};
} else {
    thisIsAStringArray = new String[] {"Asparagus", "Carrot", "Tomato"};
}
```

Note that the value of the array depends on the value of fruits variable. The array can have the values {"Apple", "Banana", "Orange"} if fruits is true, or {"Asparagus", "Carrot", "Tomato"} if fruits is false.

Note that initializing an array will override the old contents of the array. For example

```
String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};
thisIsAStringArray = new String[] {"Asparagus", "Carrot", "Tomato"};
System.out.println( thisIsAStringArray[0] );
System.out.println( thisIsAStringArray[1] );
System.out.println( thisIsAStringArray[2] );
```

The code will have the output below. This is because the old contents {"Apple", "Banana", "Orange"}, will be discarded and replaced with the new contents.

```
Asparagus
Carrot
Tomato
```

Also note that even the size of array will be changed if re-initialized. For example:

```
String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};
thisIsAStringArray = new String[] {"Asparagus", "Carrot"};
System.out.println( thisIsAStringArray[0] );
System.out.println( thisIsAStringArray[1] );
System.out.println( thisIsAStringArray[2] );
```

Will have the following output:

```
Asparagus
Carrot
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
    at ArrayTest.main(ArrayTest.java:10)
```

The exception is because the thisIsAStringArray variable will only have 2 elements after the second initialization.

### String Array Length/Size

The property length of a String Array can be used to determine the number of elements in an Array. Here is an example usage:

```
String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};
System.out.println( thisIsAStringArray.length );
```

This will have the output:

```
3
```

Because there are 3 elements in the declared array.

### Iterate Through String Array

Since the size and contents of a String Array can vary, it is useful to iterate through all the values. Here is an example code using style that can run prior to Java 5.

```
String[] thisIsAStringArray = {"Apple", "Banana", "Orange"};
for( int i = 0; i < thisIsAStringArray.length; i++)
{
    String element = thisIsAStringArray[i];
    System.out.println( element );
}
```

The code will start from index 0, and continue upto length - 1, which is the last element of the array. This code will run on any version of Java, before of after Java 5

The code will have the output:

```
Apple
Banana
Orange
```

### 3.1.2.String Methods

1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns formatted string
4	String substring(int beginIndex)	returns substring for given begin index
5	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
6	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
7	boolean equals(Object another)	checks the equality of string with object
8	boolean isEmpty()	checks if string is empty
9	String concat(String str)	concatinates specified string
10	String replace(char old, char new)	replaces all occurrences of specified char value
11	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
12	String[] split(String regex, int limit)	returns splitted string matching regex and limit
13	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
14	String toLowerCase()	returns string in lowercase.
15	String toUpperCase()	returns string in uppercase.
16	String trim()	removes beginning and ending spaces of this string.

```
// Demonstrating some String methods.
class StringDemo2 {
    public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;
        System.out.println("Length of strOb1: " +
            strOb1.length());
        System.out.println("Char at index 3 in strOb1: " +
            strOb1.charAt(3));
        if(strOb1.equals(strOb2))
            System.out.println("strOb1 == strOb2");
        else
            System.out.println("strOb1 != strOb2");
        if(strOb1.equals(strOb3))
            System.out.println("strOb1 == strOb3");
        else
            System.out.println("strOb1 != strOb3");
    }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

### 3.1.3String Buffer Class

StringBuffer is a peer class of String that provides much of the functionality of strings. As String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences.

#### StringBuffer Constructors

StringBuffer defines these four constructors:

```
StringBuffer( )
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars*.

#### length() and capacity()

The current length of a **StringBuffer** can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms:

```
int length( )
int capacity( )
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

### **setLength( )**

To set the length of the buffer within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

```
void setLength(int len)
```

Here, *len* specifies the length of the buffer. This value must be nonnegative.

### **charAt( ) and setCharAt( )**

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. The value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

For **charAt( )**, *where* specifies the index of the character being obtained. For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the buffer.

The following example demonstrates **charAt( )** and **setCharAt( )**:

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));
        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
    }
}
```

Here is the output generated by this program:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

### append()

The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

**String.valueOf()** is called for each parameter to obtain its string representation. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append()**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
    public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);
        s = sb.append("a = ").append(a).append("!").toString();
        System.out.println(s);
    }
}
```

The output of this example is shown here:  
a = 42!

### insert()

The **insert()** method inserts one string into another. It is overloaded to accept values of all the simple types, plus **Strings**, **Objects**, and **CharSequences**. Like **append()**, it calls **String.valueOf()** to obtain the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

The following sample program inserts “like” between “I” and “Java”:

```
// Demonstrate insert().
class insertDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("I Java!");
        sb.insert(2, "like ");
        System.out.println(sb);
    }
}
```

The output of this example is shown here:  
I like Java!

### reverse()

The characters can be reversed within a **StringBuffer** object using **reverse()**, shown here:

```
StringBuffer reverse()
```

This method returns the reversed object on which it was called. The following program demonstrates **reverse()**:

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}
```

Here is the output produced by the program:

```
abcdef
fedcba
```

### **delete( ) and deleteCharAt( )**

Deleting characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

The **delete( )** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*-1. The resulting **StringBuffer** object is returned.

The **deleteCharAt( )** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

Here is a program that demonstrates the **delete( )** and **deleteCharAt( )** methods:

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.delete(4, 7);
        System.out.println("After delete: " + sb);
        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
    }
}
```

The following output is produced:

```
After delete: This a test.
```

```
After deleteCharAt: his a test.
```

### **replace( )**

To replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*-1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace( )**:

```
// Demonstrate replace()
class replaceDemo {
    public static void main(String args[]) {
```

```

StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
}
}

```

Here is the output:

After replace: This was a test.

## 3.2 Classes and objects

A Java program must be encapsulated in a class that defines the state and behavior of basic program components known as objects. Classes create objects and objects use methods to communicate between them.

Classes contain a group of logically related data items (called as fields) and functions (called as methods) that operate on them. Calling a specific method in an object is described as sending the object a message.

### 3.2.1 Defining a class

A class is a user defined data type with a template that serves to define its properties. Objects are nothing but instances of classes. They can be created using declarations. The basic form of a class definition is

```

class classname [extends superclassname]
{
    [Variables declaration;]
    [Methods declaration;]
}

```

Everything inside square bracket is optional. Classname and superclassname are any valid Java identifiers. The keyword extends indicates that the properties of the super class are extended to the sub class. A class can have empty body.

Eg; - class Empty

```

{
}

```

By convention, all class names start with upper case letter.

#### 3.2.1.1 Adding variables

Data in a class is encapsulated by placing data fields inside the body of the class definition. These variables are called as instance variables and they are created whenever an object of the class is instantiated. Instance variables are declared in the same way as local variables.

Eg; - class Rectangle

```

{
    int length;
    int width;
}

```



Here, length and width are two integer type instance variables. They do not occupy any memory space. They are also called as member variables.

### 3.2.1.2 Adding methods (Defining a method)

A class can have any no. of methods called as instance methods. Methods are used to manipulate the data contained in the class. Method definitions should be present inside the class immediately after the instance variables declarations. A method is of the form

```
type method name (parameter list)
{
    method body;
}
```

where

- type can be either user defined or class type. It specifies the type of value returned by the method. The type is void, if the method doesn't return any value.
- The method name is any valid identifier name.
- The parameter list contains list of variable names and their types, which act as input to the method separated by commas. The parameter list can be empty but the parentheses are must.
- The body contains a set of statements that perform operations on the data.

Eg; - class Rectangle

```
{
    int len, wid;
    void getData (int x, int y)
    {
        len =x;
        wid =y;
    }
    int recArea ( )
    {
        int area = len * wid;
        return (area);
    }
}
```

In the parameter list, the variable names should be declared independently with types for each of them separated by commas.

Instance variables and methods in classes are accessible by all the methods in a class but a method cannot access the variables declared in other methods.

### 3.2.1.3 Creating objects (initializing an object)

An object in Java is a block of memory having enough space to store all instance variables. Objects can be created using the new operator which creates an object of the specified class and returns a reference to that object .The general form for creating an object is

```
classname object;
object = new classname( );
```

where classname is any already defined class name and object is the name of the object.

The first statement declares a variable to hold the object reference.  
The second statement assigns the object reference to the variable.

Eg; - `Rectangle rect1;`  
`rect1 = new Rectangle( );`

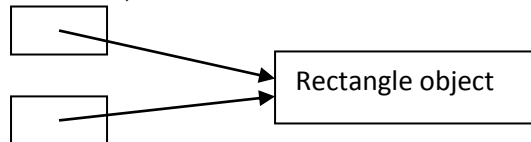
Statement	Action	Result
<code>Rectangle rect1;</code>	declare	
<code>rect1 = new Rectangle( );</code>	instantiate	

The declaration and instantiation can be combined into a single statement  
`classname object = new classname( );`

Eg;- `Rectangle rect1 = new Rectangle( );`

Any no. of objects can be created to a class  
`Rectangle rect1= new Rectangle( );`  
`Rectangle rect2= new Rectangle( );`

Any no. of references can also be created to an object  
`Rectangle rect1 = new rectangle( );`  
`Rectangle rect2 = rect1;`



### 3.2.1.4 Accessing class members

Each object has its own set of variables and methods. We cannot access these variables and methods directly from outside the class. For this purpose, the dot operator is used. Its general form is

`objectname.variablename`  
`objectname.methodname (parameter list)`

where

- `objectname` is the name of the object
- `variablename` is the name of the instance variable
- `methodname` is the name of the method
- `parameter list` is a list of actual values separated by commas.

Eg;- `rl.len =15;`  
`rl.wid = 10;`  
`r2.len =20;`  
`r2.wid =12;`

Thus, we can assign values to instance variables

R1.length	15	r2.length	20
R1.width	10	r2.width	12

We can also assign values to variables using methods declared inside the class

```
Rectangle r1 = new Rectangle( );
r1.getData(15,10);
```

R1.length	15
R1.width	10

We can compute the area

i) using assignment statement.  

```
int area1 = r1.len * r1.wid;
```

ii) by calling the rectArea( ) method declared inside the Rectangle class  

```
int area1 = r1.recArea( );
```

Eg;- class Rectangle

```
{
    int length, width;
    void getData(int x, int y)
    {
        length=x;
        width=y;
    }
    int rectArea ( )
    {
        int area = length * width;
        return (area);
    }
}

class Rectarea
{
    public static void main (String args[])
    {
        int area1, area2;
        Rectangle rect1 = new Rectangle( );
        Rectangle rect2 = new Rectangle( );
        rect1.length =15;
        rect2.width =10;
        area1 = rect1.length * rect1.width;
        rect2.getData (20, 12);
        area2 = rect2.rectArea( );
        System.out.println ("Area of rectangle 1 = " + area1);
        System.out.println ("Area of rectangle 2 = " + area2);
    }
}
```

### 3.2.2 Constructors

A constructor is a special method used to initialize an object at the time of creation. Its general form is

```
Constructor name (arguments)
{
    Initialization statements;
}
```

where constructor name is same as that of the class name. Constructors return instance of a class and hence they do not specify a return type. A constructor may or may not have arguments.

Eg:-

1) constructor without arguments

```
Rectangle ( )
```

```
{
length= 15;
width = 10;
}
```

2) Constructor with arguments

```
Rectangle (int x, int y)
```

```
{
length =x;
width = y;
}
```

eg;- class Rectangle

```
{
    int length, width;
    Rectangle (int x, int y)
    {
        length =x;
        width = y;
    }
    int rectArea( )
    {
        return (length * width);
    }
}

class RectArea
{
    public static void main (String args[ ])
    {
        Rectangle r1 = new Rectangle(15, 20);
        int area1 = r1.rectArea( );
        System.out.println("Area of rectangle =" + area1);
    }
}
```

### 3.2.3 Method overloading

Method overloading is used when objects are required to perform similar tasks but using different input parameters. All the methods have same name but they have different parameter lists and different definitions. When a method is called, java first matches the method name and then it matches the number and type of parameters to decide which one of the definitions is to be executed. This process is known as **polymorphism**.

Eg;- class Assign

```
{
    int a,b,c;
    void assignData(int x)
    {
        a=x;
    }
    void assignData(int x, int y)
    {
        a=x;
        b=y;
    }
    void assignData(int x, int y, int z)
    {
        a=x;
        b=y;
        v=z;
    }
    void print ( )
    {
        System.out.println ("a=+a);
        System.out.println ("b="+b);
        System.out.println ("c="+c);
    }
    public static void main (String args[] )
    {
        Assign obj = new Assign ( );
        Obj.assignData(1);
        Obj.print( );
        Obj.assignData(2,3);
        Obj.print( );
        Obj.assignData(4,5,6);
        Obj.print( );
    }
}
```

### 3.2.4 Static members

The static members can be declared as common to all the objects and accessed without using a particular object.

**Example:**     static int count;  
                 static float max (float x, float y);

The member that are declared as static are called static members. Static variables and static methods are also called as class variables and class methods.

**Eg:-**

```
class mathoperation
{
    static float mul (float x, float y);
    {
        return x*y;
    }
    static floatdivide (float x, float y);
    {
        return x/y;
    }
}
class mathapplication
{
    public static void main (String args[])
    {
        float a = mathoperation.mul (4.0, 5.0);
        float b = mathoperation.divide (a, 2.0);
        System.Out.println ("b="+b);
    }
}
```

**Output:**

b = 10.0

**Note:**

1. Static methods are called using the class name.
2. Static methods can only call other static methods.
3. Static methods can only access static data..
4. Static methods cannot refer to this or super in any way.

### 3.2.5 Nesting of methods

A method can be called by using only its name by another method of the same class. This is known as nesting of methods.

**Eg:-**

```
class nesting
{
    int m, n;
    nesting(int x, int y) // constructor method
    {
        m =x;
        n = y;
    }
    int largest ()
    {
        if (m>= n)
            return (m);
        else
            return (n);
    }
}
```

```

    }
    void display()
    {
        int large = largest(); // calling a method
        System.Out.println(" largest value = "+ large);
    }
}

```

#### Output:

largest value = 50

A method can call any number of methods. It is also possible for a called method to call another method.

### 3.2.6 this keyword

Sometimes a method will need to refer to the object that invoked it. **this** keyword is defined for the purpose. It can be used inside any method to refer to the current object. **this** always refers to the object on which the method was invoked. **this** can be used anywhere, where a reference to an object of the current class type is permitted.

Box (double w, double h, double d)

```

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

```

When a local variable has the same name as an instance variable, it hides the instance variable. To resolve this problem, **this** keyword can be used.

Box (double width, double height, double depth)

```

{
    this.width = width;
    this.height = height;
    this.depth = depth;
}

```

### 3.2.7 Command line arguments

If our programs have to act in a particular way depending on the input given at the time of execution, command line arguments are used. Command line arguments are parameters that are supplied to the program at the time of invoking it for execution.

Eg:-

class Comlinetest

```

{
    public static void main (String args [])
    {
        int count, i=0;
        String s;
        count =args.length;
        System.out.println ("No. of arguments =" +count);
    }
}

```

```

        while (i<count)
        {
            s=args [i];
            i=i+1;
            System.out.println (i + ". "+"java is"+s);
        }
    }
}

```

Compile using  
 javac comlinetest.java

Run using  
 java comlinetest simple object\_oriented distributed robust secure

The command line arguments are passed to the program through the array args. The command line has 5 arguments. They are assigned to the array args as follows

```

args[0] ="simple"
args[1] = "object_oriented"
args[2] = "distributed"
args[3] = "robust"
args[4] = "secure"

```

### Output

No. of arguments =5

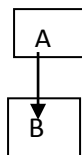
1. java is simple
2. java is object oriented
3. java is distributed
4. java is robust
5. java is secure

## 3.3 Inheritance

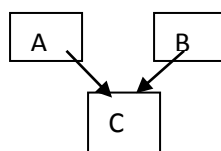
The process of deriving a new class form an existing one is called as **inheritance**. The old class is called as the **base class or super class or the parent class** and the new class is known as **derived class or subclass or child class**. The subclass inherits all the variables and methods of the super class plus its own variables and methods.

### 3.3.1 Types of inheritance

- a) Single inheritance – A subclass is derived from only one super class.



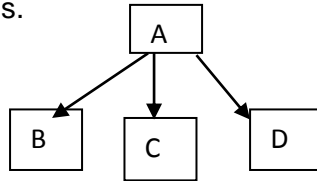
- b) Multiple inheritances – A subclass is derived from more than one super class



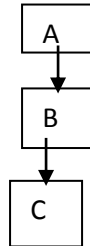


Java doesn't support multiple inheritance. This concept is implemented using a secondary inheritance path in the form of interfaces.

c) Hierarchical inheritance – More than one subclass is derived from a super class. This can be extended to any no. of levels.



D) Multilevel inheritance - A sub class is derived from another derived class.



### 3.3.2 Defining a subclass

A subclass is defined as follows

```
class subclassname extends superclassname
{
    variables declaration;
    methods declaration;
}
```

The keyword extends signifies that the properties of the super class are extended to the subclass. The subclass will have its own variables and methods as well as that of the super class.

Eg:-

```
class Rect
{
    int l,b;
    Rect (int x, int y)
    {
        l=x;
        b=y;
    }
    int area ( )
    {
        return (l*b);
    }
}
class Box extends Rect
{
    int h;
    Box (int a, int b, int ( )
    {
```

```

        super (a, b);
        h=c;
    }
    int vol ( )
    {
        return (l*b*h);
    }
}
class Volume
{
    public static void main (String args[ ])
    {
        Box b1 = new Box 10,10,10);
        int a = b1.area( );
        int v = b1.vol( );
        System.out.println("Area =" +a);
        System.out.println("Volume=" +v);
    }
}

```

The program defines a class Rect and extends it to another class box. The constructor in the derived class uses the **super** keyword to pass values that are required by the base class constructor.

### 3.3.2.1 Subclass constructor

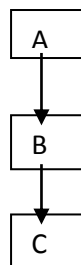
It is used to construct the instance variables of both the sub class and the super class. It uses the keyword **super** to invoke the constructor of the super class.

```
super(argument list);
```

The super keyword can only be used inside a subclass, and it must be the first statement within the subclass. The argument list should match the order and type of instance variables declared in the super class.

### 3.3.3 Multilevel inheritance

This concept allows us to build a chain of classes



The class A serves as a base class for the derived class B which in turn serves as the base class for the derived class C .The chain ABC is known as **inheritance path**.

```

class Line
{
    int l;

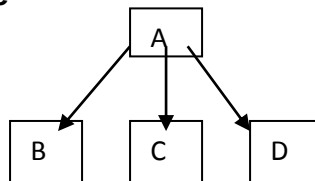
```

```

        Line (int x);
        {
            l =x;
        }
    }
class Square extends Line
{
    Square (int y)
    {
        super (y);
    }
    int area ( )
    {
        return l*l;
    }
}
class Cube extends Square
{
    Cube (int z)
    {
        super (z);
    }
    int vol ( )
    {
        return l*l*l;
    }
}
class Main ( )
{
    public static void main (String args[ ])
    {
        Cube c1 = new Cube(5);
        System.out.println ("Area=" c1.area );
        System.out.println ("Volume=" c1.vol( ));
    }
}

```

### 3.3.4 Hierarchical inheritance



It is the process of deriving classes in a hierarchical manner. The features of one level are shared by many others below the level. The data items that are common are put into the super class. The classes at the lower level possess all the properties of its superior classes.

```

class Polygon
{
    int noofside;
    Polygon(int s)
    {

```

```

        noofside =s;
    }
}
class Triangle extends Polygon
{
    intb,h;
Triangle(int x, int y)
{
    super(3);
    b = x;
    h =y;
}
    int area ( )
    {
        return b*h*2;
    }
}
class Square extends Polygon
{
    int s;
Rectangle (int a)
{
    super (4);
    s = a;
}
    int area ( )
    {
        return s*s;
    }
}
class Hierarchy
{
    public staic void main (String args[ ])
    {
        Triangle T = new Triangle(10,5);
        System.out.println("Triangle");
        System.out.println("No. of sides =" noofside);
        System.out.println("Area =" +T.area ( ));
        Square R = new Square(5);
        System.out.println("Rectangle");
        System.out.println ("No.of sides =" noofside);
        System.out.println("Area=" +R.area ( ));
    }
}

```

### 3.3.4 Overriding methods

Suppose that an already defined method in a super class is redefined with the same name, arguments and return type as a method in the subclass. When the method is called, then the method defined in the subclass is invoked and executed and the super class method is neglected. This concept is known as **overriding**.

```

class Super1
{
    int x;
    Super1 (int a)
    {
        x = a;
    }
    void disp( )
    {
        System.out.println ("x"= +x);
    }
}
class Sub extends super1
{
    int y;
    Sub (int a, int b)
    {
        super (a);
        y = b;
    }
    void disp( )
    {
        System.out.println ("y = " + y);
    }
}
class Test
{
    public static void main (String args[ ])
    {
        Sub s1 =new Sub(10,20);
        s1.disp( );
    }
}

```

### Output

y =20

If the method in the super class is to be invoked, then the syntax is

super.superclass method ( )

Now, modify the disp( ) method in the class Sub as follows

```

void disp ( )
{
    super.disp( );
    System.out.println ("y=" +y);
}

```

### Output

x = 10

y = 20

### 3.3.5 Final variables and methods

By default, all the methods and variables in super classes can be overridden. If we want to prevent the subclass from overriding the members of the super class, they can be declared using the modifier final. If a method or variable is declared as final, it can never be changed. If we try to change, the compiler generates an error message.

```
final data type variable = literal;
final returntype functionname (parameterlist)
{
    method body;
}
```

eg:-

```
class Super1
{
    final int x;
    Super1 (int a)
    {
        x = a;
    }
    final void disp( )
    {
        System.out.println ("x=" +x);
    }
}
class Sub extends Super1
{
    int y;
    Sub1 (int a, int b)
    {
        super (a);
        y = b;
    }
    void print ( )
    {
        super.disp( )
        System.out.println ("y=" +y);
    }
}
class Maintest
{
    public static void main (String args[ ])
    {
        Sub1 s1 = new Sub1(5,10);
        s1.print ( );
    }
}
```

#### Output

```
x = 5
y = 10
```

### 3.3.5.1 Final classes

A final class is defined with the modifier **final**. If a class is declared as **final**, then all its variables and methods are final and hence it cannot be inherited. We cannot derive subclasses for a final class. Suppose that we have a final class named xxx

```
final class xxx
{
    .....
}
```

Then it is impossible to define a new sub class. Trying to do so will produce an error message.

eg:- class yyy extends xxx

```
{
    .....
}
```

The above class produces an error message.

### 3.3.6 Abstract methods and classes

The modifier **abstract** when used in method definition indicates that the method must always be redefined in a subclass, thus making overriding compulsory. Its general form is

```
abstract returntype methodname (parameterlist)
{
    .....
}
```

When a class contains one or more abstract methods, then it should also be defined as abstract. Its general form is

```
abstract class classname
{
    .....
}
```

Abstract classes cannot be used to instantiate objects directly. The abstract methods of an abstract class should be redefined in its sub class. We cannot create abstract constructors or abstract static methods.

Eg:-

```
abstract class First
```

```
{
    int x;
    abstract void disp( );
}
```

```
class Second extends first
```

```
{
    int y;
    void disp( )
    {
        System.out.println ("x = " + x);
        System.out.println ("y = " + y);
    }
}
```

```

    }
class Demoabs
{
    public static void main (String args[ ])
    {
        Second obj = new Second( );
        obj.x =10;
        obj.y =20;
        obj.disp( );
    }
}

```

### Output

```

x = 10
y = 20

```

### 3.3.7 Visibility control

Restricting the access of certain variables and methods from outside the class is some times unavoidable. For this visibility modifiers are applied to the instance variables and methods. The visibility modifiers are also known as access modifiers. There are five types of visibility modifiers. They are

1. public
2. private
3. protected
4. friendly
5. private protected

#### 3.3.7.1 Public access

A variable or method declared as public can be accessed from everywhere.

#### Syntax:

```

public datatype variablename;
public returtype methodname()
{
    ----
}

```

**Example:**

```

public int x;
public void sum()
{
}

```

#### 3.3.7.2 Friendly access (Package public)

A variable or method declared as without any access specifier becomes friendly or package public. It can be accessed by all program in same package. It is the default modifier.



## Syntax

```
datatype variablename;  
returntype methodname()  
{  
    ----  
}
```

### Example:

```
float y;  
int add (int x, int y)  
{  
    ----  
}
```

### 3.3.7.3 Protected access

A variable or method declared as protected are visible everywhere in the current package and also sub classes in other packages

#### Example

```
protected int x;  
protected void add()  
{  
    ----  
}
```

### 3.3.7.4 Private access

A variable or method declared as private are visible only to its own class.

#### Example

```
private int x;  
private void add()  
{  
    ----  
}
```


### 3.3.7.5 Private protected

A variable or method declared as private protected are visible only in sub class in all the packages.

#### Example

```
private protected int x;  
private protected void add()  
{  
    ----  
}
```

Access Modifier Access Location	Public	Protected	Friendly	Private
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No
Subclasses in other package	Yes	Yes	No	No
Non subclasses in other package	Yes	No	No	No



### 3.3.8 Interfaces

Java doesn't support multiple inheritance. Multiple inheritance can be achieved through interfaces. Though a java class cannot be a sub class of more than one super class, it can implement more than one interface, thereby enabling us to create classes that build upon other classes.

#### 3.3.8.1 Defining interfaces

An interface is basically a kind of class. An interface can define only abstract methods and final fields. Hence an interface does not specify any code to implement these methods and data fields contain only constants. Hence, the class that implements an interface should define that code for the implementation of these methods.

```

The general form of an interface definition is
interface interfacename
{
    Variables declaration;
    Methods declaration;
}

```

The interface name can be any valid Java variable name. The variables are declared as follows

```
Static final datatype variable = value;
```

All variables are assigned constant values.

Methods declaration will contain only a list of methods without any body.

#### Syntax

```
returntype methodname (parameterlist);
```

Eg;- interface Area

```
{
    final static float PI =3.14f;
    float compute (float x, float y)
    void show ( );
}
```

### 3.3.8.2 Extending interface

Interfaces can be extended i.e one interface can be derived from the other one. This is achieved with the help of **extends** keyword.

interface interfacename extends interfacename 1, interfacename 2 .....

```
{
    .....
}
```

eg;- interface Itemconstants

```
{
    int code = 1001;
    string name = "Fan";
}
interface ItemMethods
{
    void disp( );
}
interface Item extends ItemConstants, ItemMethods
{
    .....
}
```

The sub interfaces cannot define the methods declared in the super interfaces. All the methods should be defined only in the class that implements the derived interface.

### 3.3.8.3 Implementing interfaces

Interfaces are used as super classes whose properties are inherited by classes. Hence, it is necessary to create a class that inherits the interface.

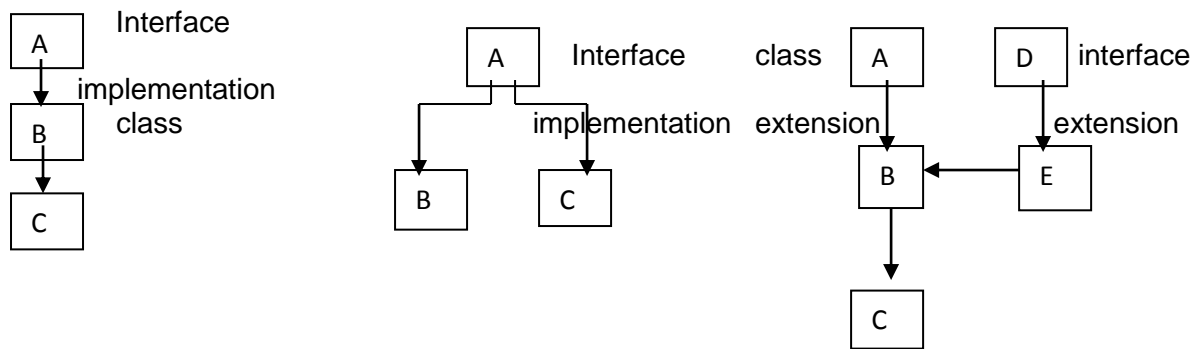
class classname implements interfacename

```
{
    .....
}
```

class classname extends superclassname implements interfacename1,  
interface name 2.....

```
{
    .....
}
```

The implementation of interface can take various forms



Eg;-

interface Area

```
{
    final static float pi = 3.14f;
    float compArea (float x, float .y);
}
```

class Rectangle implements Area

```
{
    public float compArea(float x, float y)
    {
        return (x*y);
    }
}
```

class Circle implements Area

```
{
    public float compArea (float x, float y)
    {
        return (pi*x*x);
    }
}
```

class Calcarearea

```
{
    public static void main (string args [ ])
    {
        Rectangle r = new Rectangle ( );
        Circle c = new Circle ( );
        Area a;
        a = r;
        System.out.println ("Area of rectangle =" + a.compArea(10,20));
        a = c;
        System.out.println ("Area of circle =" + a.compArea(10,0);
    }
}
```

**Output**

Area of Rectangle = 200

Area of circle = 314

If a class that implements an interface, doesn't implement all the methods of the interface, the class becomes an abstract class and hence cannot be instantiated.

#### 3.3.8.4 Accessing interface 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. Such interfaces do not contain methods. The constant values are available to any class that implements the interface. The values can be used in any method, as part of variable declaration or anywhere where we can use a final value.

Eg:-

```
interface Cricket
{
    int noofplayers = 11,
    int noof keepers = 1;
}
class Game implements Cricket
{
    void print ( )
    {
        System.out.println ("No.of players =" +noofplayers);
        System.out.println ("No.of keepers = " + noof keepers);
    }
}
class Header
{
    public static void main (String args [ ])
    {
        Game g = new Game ( );
        g.print ( );
    }
}
```

#### Output

```
No. of players = 11
No. of keepers = 1
```

#### 3.3.8.5 Implementing multiple inheritance through interface

```
class Stud
{
    int rollno;
    void getNo(int n)
    {
        rollno = n;
    }
    void putNo( )
    {
        System.out.println ("Roll No. =" + rollno);
    }
}
```

```

class Test extends Student
{
    int m1, m2, m3;
    void getMarks(int x, int y, int z)
    {
        m1 = x;
        m2 = y;
        m3 = z;
    }
    void putMarks( )
    {
        System.out.println("Mark 1 =" + m1);
        System.out.println("Mark 2 =" +m2);
        System.out.println("Mark 3 =" +m3);
    }
}
interface Sports
{
    int bonus = 100;
    void showBonus ( );
}
Class Result extends Test implements Sports
{
    int total;
    public void showBonus( )
    {
        System.out.println ("Bonus marks =" +bonus);
    }
    void disp( )
    {
        total = m1 +m2 + m3 + bonus;
        putNo( );
        putMarks ( );
        showBonus ( );
        System.out.println ("Total Score =" total);
    }
}
class Hybrid
{
    public static void main (String args[ ])
    {
        Result s1 = new Result( );
        S1.getNo(98430);
        S1.getMarks(99,73,87);
        S1.disp( );
    }
}

```

## Review Questions

### Part A (2 marks)

1. Define classes and objects.
2. What are constructors?
3. What is the use of this keyword?
4. What is inheritance?
5. List the different access modifiers.
6. What are interfaces?

### Part B (3 marks)

1. How will you define a class.
2. What are static members?
3. What are command line arguments?
4. What are the types of inheritance?
5. How will you define a subclass?
6. What are final variables and methods?
7. What do you mean by abstract classes?
8. How will you define an interface?

### Part C (5 marks)

1. Explain how will you create a class with an example.
2. Explain constructors with example.
3. Explain method overloading with example.
4. Explain any one type of inheritance with example.
5. Explain the different types of access modifiers?
6. Explain how will you implement multiple inheritance through interface?

## **Unit IV- PACKAGES, APPLET, AWT AND EVENT HANDLERS**

### **Objectives**

- To create and access packages
- To write simple applets
- To list the types of AWT components
- To study the various event handlers

### **4.1 Packages**

Packages act as “containers” for classes. They provide a functional way of grouping classes and /or interfaces together. Packages help us to reuse the code already created.

### **Benefits**

- We can easily reuse the classes contained in the packages of other programs.
- Two classes in two different packages can have the same name. To access a particular class, we give the package name and the class name separated by a dot.
- Packages hide their classes from other programs and other packages
- Packages also provide a way for separating “design” from “coding”. First, we can design the classes, decide their relationships and then we can implement the code needed for the methods. Hence, the implementation of any method can be changed without affecting the rest of the design.

### **Classification**

Java packages are classified into

- System packages or JSL (Java Standard Library) or API Packages (Application Programmers Interface) – defined by the manufacturers.
- User defined packages – defined by the user.

#### **4.1.1 System packages**

The frequently used system packages are

##### **1) java.lang**

This is the default package that is automatically imported and it contains classes that support the java language including primitive types, strings, Math functions, Threads and exceptions.

##### **2) java.util**

This contains classes for utilities such as vectors, hash tables, random nos., date etc.

##### **3) java.io**

This contains classes for supporting input and output operations.

##### **4) java.awt**

This contains classes for implementing GUI elements such as windows, buttons, lists, menus.

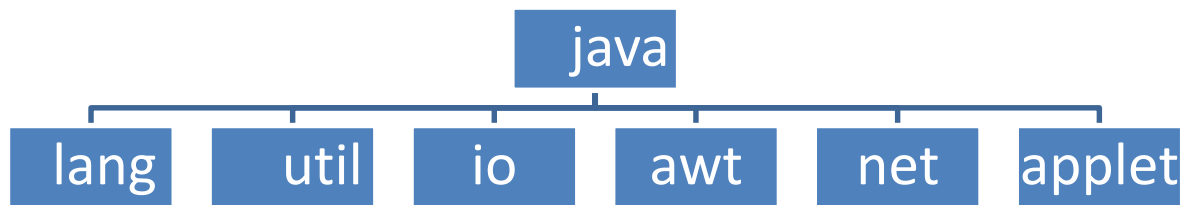
##### **5) java.net**

This contains classes for networking with local computers as well as with the internet servers.

##### **6) java.applet**

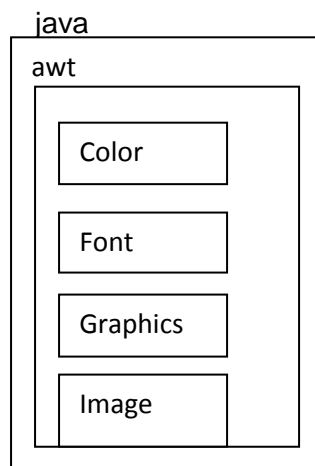
This contains classes for creating and implementing the applets.





#### 4.1.1.1 Using System packages

The packages are organized in a hierarchical structure. For example, the java package contains the awt package which in turn contains classes for implementing GUI.



There are two ways of accessing the classes stored in a package

1. By using fully qualified class name

This is achieved by using the package name containing the class and then appending the class name to it using the dot operator.

Eg;- `java.awt.font`

refers to the class font in the awt package which is in turn contained in the java package

2. Using import statement

This is used when either one or more classes belonging to a package are used at one or more places in a program.

#### **Syntax**

`import package name.package name. .... class name;`

(Or)

`import package name.*;`

The import statement appears before the class definition

Eg; - 1) import java.awt.color:  
                imports the class color  
       2) import java.awt.\*;  
imports all the classes in the awt package.

#### 4.1.1.2 Naming conventions

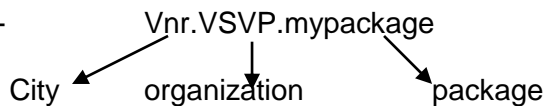
Any standard identifier can be used as a package name. By convention, all package names start with lowercase letters.

Eg;-

```
double y=java.lang.Math.sqrt (x) ;
```

Package names must be unique. Duplicate names cause runtime errors. Since, java programmers work on Internet, duplicate package names are unavoidable. Hence, the java designers suggest the users to use the domain names as prefix to the package names.

Eg;-



### 4.1.2 Creating packages

1. Create a subdirectory with the same name as that of the package name inside the directory where classes that will import the package to be created are stored.
2. Declare the package using the package statement. This must be the first statement in a Java source file.

```
package packagename;
```

3. Define new classes for the package. A package file can have more than one class definition. Declare any one class as public.

```
public class class name;
```

$$\left\{ \begin{array}{c} \vdots \\ \vdots \end{array} \right\}$$

```
class class name 2
```

$$\left\{ \begin{array}{c} \vdots \\ \vdots \end{array} \right\}$$

4. Store the file with the name same as that of the class name declared as public and with extension .java inside the directory with the package name.
5. Compile the file so that class files with class extension are created for each class definition in a package

Java also supports the concept of package hierarchy. This is done by specifying multiple package names separated by dots in a package statement

```
package package name1.package name2;
```

This helps us to group related classes into a package and then group related packages into a package and then group related packages into a larger one. The new package should be stored in the subdirectory with package name2, which is in turn the subdirectory of directory with package name1, which is in turn the subdirectory of the directory where the source files that will import the package are stored.

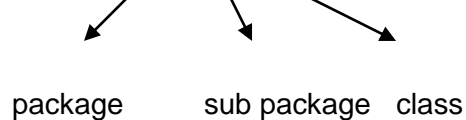
### 4.1.3 Accessing a user package

#### 1, Using fully qualified class name

A package can be accessed by specifying the class path as

Package name1.package name 2. .... .class name

Eg;- college.student.Print



This will access the class Print in the sub package student which in turn belongs to the package college.

#### 2. Using import statement (short cut approach)

This statement can be used to search a list of packages for a particular class.

import package name 1.[package name 2. ....].class name;

There can be more than one import statement and it should appear before the class definitions. After an import statement is defined, all the members can be directly accessed without using the package name or class name

Eg;- 1. import student.Result;

Allows us to use the class Result in the package student.

2. import student.\*;

Allows us to use all the classes defined in the package student.

3. import college.student.Result;

Allows us to use the class Result in the sub package student in the package college.

The short cut approach has the advantage that we need not use long package names repeatedly in our programs. But it has the disadvantage that it is difficult to determine from which package a particular member came, when a large no. of packages are imported.

#### 4.1.4 Using a package

```
package college;
public class Student
{
    int rollno;
    String name;
    public Student (int r, string n)
    {
        rollno = r;
        name = n;
    }
    public void print ( )
    {
        System.out.println ("Roll No ." + roll no);
        System.out.println ("Name:" + name);
    }
}
```

Store the file in the subdirectory college with the name Student.java. Compile the file and store the class files in the same directory.

```
package course;
public class Diploma
{
    String discipline;
    String year;
    public Diploma(String d,String y)
    {
        discipline = d;
        year = y;
    }
    public void print ( )
    {
        System.out.println ("course:" + discipline);
        System.out.println ("year:" + year);
    }
}
```

Store the file in the subdirectory course with the name Diploma.java. Compile the file and store the class file in the same directory.

```
import college.*;
import course.*;
class Stuinfo
{
    public static void main (String args[])
    {
        Student s1 = new Student ("98630", "kumar");
        Diploma d1 = new Diploma ("Computer", "III");
        s1.print( );
        d1.print( );
    }
}
```

## Output

Roll No : 98630  
Name : Kumar  
Course : Computer  
Year : III

### 4.1.5 Adding a class to a package

1. Place the package statement before class definitions
  2. Define the class with public modifier
- ```
package    package name ;  
public class class name  
{  
.....  
}
```
3. Save the source file with class name.java extension inside the directory with given package name.
  4. Compile the source file to create the class file which should also reside in the directory with given package name.

Any no. of classes can be added to the same package

```
class class name  
{  
    .....  
}
```

To create a package with multiple public classes, create them as separate source files with the package statement at the top of each file and compile them individually to produce the class files.

### 4.1.6 Hiding classes

Whenever a package is imported using \* option, only public classes are imported. Other classes which are not declared as public are hidden from access from outside the package.

Eg;-

```
package pl;  
public class xxx  
{  
    .....  
}  
class yyy  
{  
    ....  
}
```

Consider the section of code in the main program

```
import pl.*;  
xxx objx;  
yyy objy;
```

produces an error since yyy is a nonpublic class. It cannot be imported. It is hidden. So we can't create object for it.

## 4.2 Applets

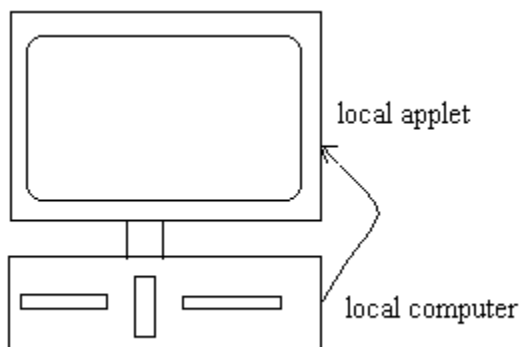
### 4.2.1 Introduction

Applets are small Java programs that are primarily used in Internet computing. They can be transported over the Internet from one computer to another System. It can be run using the Applet viewer or any web browser that supports Java. An applet can perform arithmetic operations, display graphics, play sounds, accept user input, create animation and play interactive games. A web page can now contain not only a simple text or a static image but also a Java applet which when run, can produce graphics, sounds and moving images

#### 4.2.1.1 Local and remote applets

##### 1. Local applet

They are developed locally and stored in a local System. Internet connection is not required. When a web page is trying to find a local applet, it simply searches the directories in the local System and locates and loads the specified applet.



##### 2. Remote applets

A remote applet is the one developed by the remote user and stored on a remote computer connected to the Internet. A remote applet can be downloaded on to our System through the Internet and executed.

To locate and load a remote applet, the applets address should be known. This address is known as Uniform Resource Locator (URL) and must be specified in the applets HTML document as the value of the CODEBASE attribute.

Eg;-

CODEBASE = [http: // www.netserve.com/applets](http://www.netserve.com/applets)

If the applet is local, CODEBASE may be absent or may specify a local directory.

#### 4.2.1.2 Differences between applets and application programs

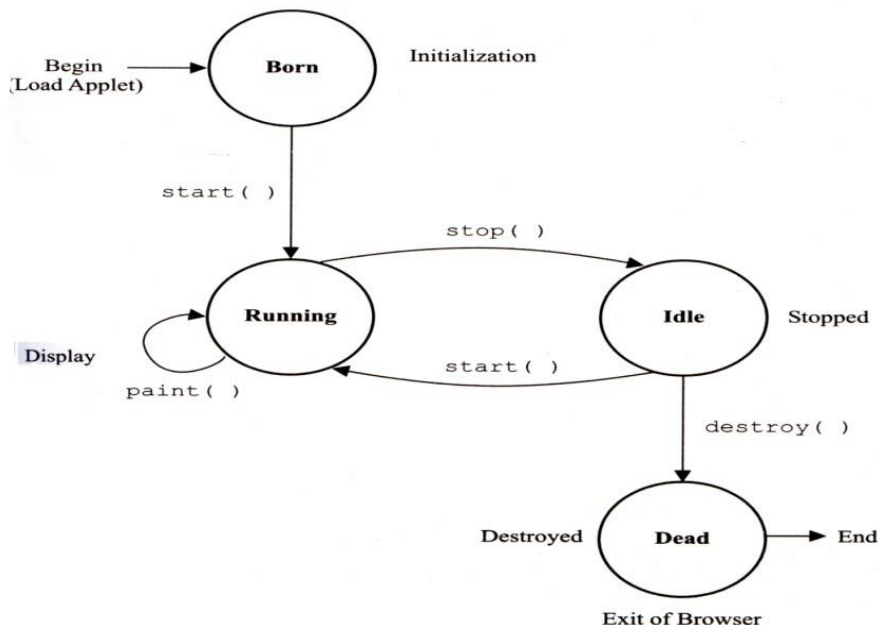
Applets are not full featured application programs. They are usually written to accomplish a small task or a component of a task. They are usually designed for use on the Internet.

1. Applets do not have main() method for initiating execution of the code. Applets when loaded automatically call certain methods of applet class to start and execute the applet code.
2. Applets cannot run independently. They are run from inside a web page using a special feature known as HTML tag.
3. Applets cannot read from or write to the files in the local computer.
4. Applets cannot communicate with other servers on the network.
5. Applets cannot run any program from the local computer.
6. Applets are restricted from using libraries from other languages such as C or C++.
7. Applets are event driven whereas the applications are control driven.

#### 4.2.2 Applet life cycle

Whenever an applet is loaded, it undergoes a series of changes in its state. The applet states include

1. Born or initialization state
2. Running state
3. Idle or stopped state
4. Dead or destroyed state



##### 1. Initialization state

Whenever an applet is loaded, it enters into initialization state. This is achieved by calling the init() method of the Applet class. The initialization occurs only once in the applets life cycle and it results in the birth of the applet. At this stage, we can

- a. create objects needed by the applet
- b. setup initial values
- c. load images or fonts
- d. setup colors

The init() method should be overridden in our program to achieve the same.

```

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

## 2. Running state

When the start() method of the Applet Class is called, an applet enters into the running state. This occurs automatically after the applet is initialized. An applet can also be restarted from idle or stopped state. The start() method can be overridden to create a thread to control the applet

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

## 3. Display State

An applet can enter display state at any time from the running state. At this state, it perform some output operation on the screen by calling the paint ( ) method. Hence, the paint ( ) method should be overridden to display information.

```
public void paint (Graphics g)
{
    .....
}
```

## 4. Idle or stopped state

An applet is stopped from running when stop() method is called and it becomes idle. Stopping occurs automatically when we leave the page containing the currently running applet.

```
public void stop()
{
    .....
}
```

## 5. Dead state

When an applet is removed from memory, it enters into the dead state. When we quit the browser, an applet automatically becomes dead by invoking the destroy() method. Dead state occurs only once in the applets life cycle. If the applet has created any resources like threads, they should be cleaned up by overriding the code.

```
public void destroy()
{
    .....
}
```

### 4.2.3.1 Creating an applet

The 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 and provides life and behavior to the applet through its methods. The Applet class therefore maintains the life cycle of an applet.

### Methods

#### 1. init( )

It is used to initialize the variables. This is the first method called and is called only once during the runtime.

```
void init ( )
```

#### 2. start ( )

This method is called by the browser after the init() method is called. It is also used to restart an applet after it has been stopped.

```
void start ( )
```

#### 3. stop( )

This method is used to suspend the execution of the applet. The execution can be restarted at any time using start() method.



```
void stop( )
```

4. `destroy ( )`

This method removes the applet completely from the memory. Before using this method, the applet should be stopped with the `stop ( )` method.

```
void destroy( )
```

5. `paint ( )`

This method is used to display the result of the applet code on the screen. The output may contain text, graphics or sound. This method can be called any no. of times and it requires an instance of a `Graphics` class as its argument.

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

Hence, the `graphics` class contained in the `java.awt` package should be imported. The `graphics` class contains a method `drawString ( )` to display the graphical applet o/p.

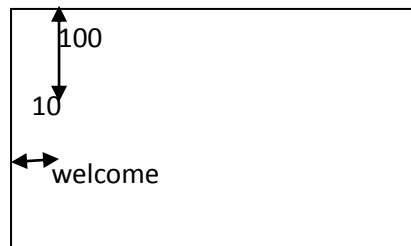
```
g.drawString(string, x, y)
```

where `string` denotes any given string and `x` and `y` denote the starting location of the screen (in pixels) where the output is drawn.

Eg;-

```
g.drawString ("Welcome", 10,100);
```

### Output



The general form for creating an applet code is

```
import java.awt.*;
```

```
import java.applet.*;
```

```
.....
```

```
public class appletclassname extends Applet
{
    public void paint (Graphics g)
    {
        .....
    }
}
```

The `appletclassname` is the main class for the applet. When the applet is loaded, Java creates an instance of this class and then calls a series of `Applet` class methods to execute the code. The main `Applet` class should be declared as `public` and the applet code should be saved with the file name of the class declared as `public` followed by `.java` extension.

Eg;-

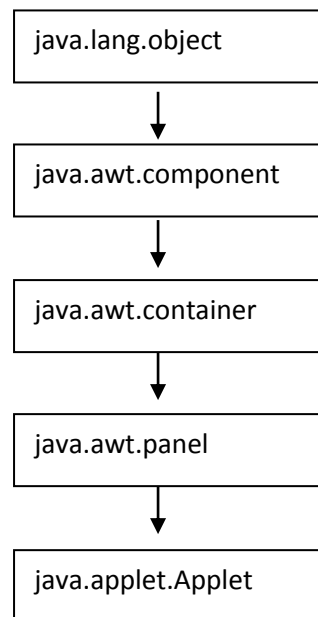
```
import java.awt.*;
```

```
import java.applet.*;
```

```
public class HelloJava extends Applet
```

```
{
    public void paint (graphics g)
    {
        g.drawString ("Hello! Welcome", 10,100);
    }
}
```

The Applet class contained in the java.applet package itself is a subclass of the panel class in java.awt package which is again a subclass of the container class, a subclass of the component class, a subclass of the object class in the java.lang package. Hence, the main Applet class inherits all the above classes i.e we can use the variables and methods from all the above classes.



#### 4.2.3.2 Executing an applet

Executable applet is nothing but the class file of the applet which is obtained by compiling the source code using the command

```
javac filename.java
```

The output will have the name filename.class and is stored in the same directory where the .java file is stored. During compilation, if any errors occur, then they should be corrected and the source file is to be recompiled.

**Note:** Make sure that the applet source code file with extension .java, executable applet file with extension .class and HTML file with extension .HTML are stored in the current directory.

An applet can be run either using

- a) Web browser supporting Java (Any Java enabled web browser)
- b) Java appletviewer.

If web browser is used, the entire web page containing the applet will be displayed.

If an appletviewer is used, then only the o/p of the applet is displayed.

To run using applet viewer, type at the dos prompt  
appletviewer HTML filename

Eg:-

appletviewer HelloJava.HTML

## Output

|                         |
|-------------------------|
| appletviewer:Hello.java |
| Applet                  |
| Hello java              |

To run using web browser, start the browser, give the file name as

C:\directory\HTML file

Eg;- C:\Java\ HelloJava.HTML

## Output

[illegible]

#### 4.2.4 APPLET tag

The `APPLET` tag is contained in the body section and starts with `<APPLET>` and ends with `</APPLET>`. It contains the name of the applet to be loaded. It tells the browser about the space requirements of the applet.

```
<APPLET
    CODE = Name of the applet
    WIDTH = Applet width
    HEIGHT = Applet height>
```

```
</APPLET>
EG;- <APPLET
      CODE = HelloJava.class
      WIDTH = 400
```

```
        HEIGHT = 200>
<APPLET>
```

The above section of code tells the browser to load the applet HelloJava.class and display it in an area with 400 pixels width and 200 pixels height.

#### 4.2.4.1 Adding Applet to the HTML file

The <APPLET> tag is inserted in the HTML page at the place where the output must appear.

Eg:-

```
<HTML>
  <! Display a welcome title and message>
  <HEAD>
    <TITLE>
      welcome to JAVA Applets
    </TITLE>
  </HEAD>
  <BODY>
    <CENTRE>
      <H1>
        welcome to the world of applets
      </H1>
    </CENTRE>
    <BR>
    <CENTER>
      <APPLET
        CODE = HelloJava.class
        WIDTH = 400
        HEIGHT = 200>
      </APPLET>
    </CENTER>
  </BODY>
</HTML>
```

The file is stored with the name HelloJava.HTML in the directory where the executable applet is stored.

#### 4.2.4.2 More about Applet tag

##### Syntax

```
<APPLET
  [CODEBASE = code base-VRL]
  CODE = applet file name.class
  [ALT = alternate text]
  [NAME = applet instance name]
  WIDTH = No. of pixels
  HEIGHT = No. of pixels
  [ALIGN = alignment]
  [VSPACE = no. of pixels]
  [HSPACE = no. of pixels]
>
[<PARAM NAME = NAME 1  VALUE = value 1 >]
[<PARAM NAME = NAME 2  VALUE = value 2 >]
.....
```

[Text to be displayed in the absence of Java]

</APPLET>

#### List of attributes

1. **CODE** It specifies the name of the applet class to be loaded.

2. **CODE BASE**

It specifies the directory in which the applet resides. If the applet resides in the same directory as the HTML file, the CODEBASE need not be present. If the applet is local and resides in different directory, then the CODEBASE should contain the path of the directory where the executable applet resides. If the applet is remote and resides in a distant computer, the CODEBASE contains the URL of the directory where the executable applet resides.

3. **WIDTH**

It specifies the width of the applet in no. of pixels.

4. **HEIGHT**

It specifies the height of the applet in no. of pixels.

5. **NAME**

It specifies a name for the applet so that the other applet on the page can refer to this applet for inter applet communication.

6. **ALIGN**

Specifies the location on the HTML page where the applet will appear. We can align them at TOP, BOTTOM, LEFT, RIGHT, MIDDLE, ABSMIDDLE, ABSBOTTOM, TEXTTOP and BASELINE.

6. **HSPACE**

Specifies the amount of horizontal blank space in no. of pixels, the browser should leave surrounding the applet (Used only when ALIGN is set to LEFT OR RIGHT).

7. **VSPACE**

Specifies the amount of vertical blank space in no of pixels, the browser should leave surrounding the applet (Used only when ALIGN is set to TOP or BOTTOM).

8. **ALT**

Specifies the text to be displayed in non Java. The attributes CODE, WIDTH and HEIGHT must compulsorily be present in an applet. Others are optional.

#### 4.2.5 Passing parameters to applets

<PARAM .....> tags can be used to pass user defined parameters to an applet. Each <PARAM .....> tag has a name attribute and a value attribute. Inside the applet code, the applet can refer to that parameter by name to find its value.

Eg;-

```
<PARAM NAME = text VALUE = " I LOVE INDIA">
```

Changes the text to be displayed

```
<PARAM NAME = color VALUE = "red">
```

Changes the color of the text to be displayed as red.

To setup and handle parameters

1. Include appropriate <PARAM ....> tags in the HTML document.
2. Provide code in the applet to parse these parameters

When an applet is loaded, parameters are passed to it. The init( ) method can be defined to get hold of the parameters in the <PARAM> tags using getParameter( ) method. The method takes one string argument representing the name and returns a string containing the value of that parameter.

Eg;-

```
import java.awt.*;
```

```
import java. applet.*;
```

```
public class HelloJavaParam extends Applet
```

```

{
    String str;
    public void init ( )
    {
        str = getParameter ("String");
        if (str == NULL)
            str = "JAVA";
        str = "Hello" + Str;
    }
    public void paint (Graphics g)
    {
        g.drawString (str,50,50);
    }
}

<HTML>
    <HEAD>
        <TITLE>
            welcome to Java applets
        </TITLE>
    </HEAD>
    <BODY>
        <APPLET
            CODE = HelloJavaParam.class
            WIDTH = 400
            HEIGHT = 200
        >
        <PARAM NAME = "String" VALUE = "APPLET !" >
    </APPLET>
    </BODY>
</HTML>

```

### Output

Hello APPLET !

If the <PARAM ....> tag is omitted, then

### Output

Hello Java

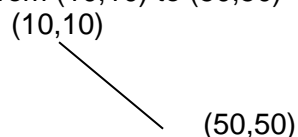
## 4.2.6 Graphics class

The class has methods for drawing many different types of shapes, from simple lines to polygons to text in a variety of fonts. These methods have arguments representing end points, corners or starting locations of a shape as values in the applets coordinate System.

### 4.2.6.1 Lines and Rectangles

The simplest shape is a line. The drawLine() method takes two coordinate pairs (x1,y1) and (x2,y2) as arguments and draws a line between them.

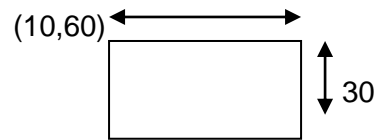
Eg;- g.drawLine(10,10,50,50);  
 Draws a line from (10,10) to (50,50)



G is the Graphics object passed to paint() method.

The drawRect ( ) method is used to draw a rectangle. It takes 4 arguments. The first two represent the x&y coordinates of the top left corner of the rectangle and the remaining two the width and height of the rectangle.

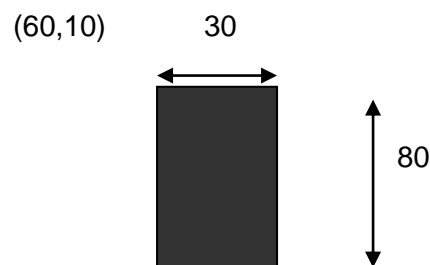
Eg;- g.drawRect(10,60,40,30);



Draws a rectangle starting at (10, 60) having a width of 40 pixels and a height of 30 pixels.

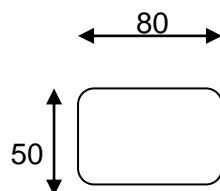
The fillRect( ) method is used to draw a filled rectangle. It also takes 4 arguments as drawRect( ).

g.fillRect(60,10,30,80);



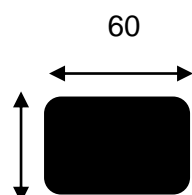
The drawRoundRect( ) method draws a rounded rectangle. It takes 6 arguments. The first four are similar to that of the drawRect( ) method. The other two represent the width & height of the corner angles.

g.drawRoundRect(10,100,80,50,50,10,10):



The fillRoundRect( ) method draws a filled rounded rectangle. The arguments are same as that for drawRoundedRect( ).

g.fillRoundRect(20,110,60,30,5,5);



```

import java.awt.*;
import java.applet.*;
public class LineRect extends Applet
{
    public void paint (Graphics g)
    {
        g.drawRect(10,100,80,50);
        g.fillRoundRect (20,110,60,30,5,5);
        g.drawLine(10,100,90,150);
        g.drawLine(10,150,90,100);
    }
}

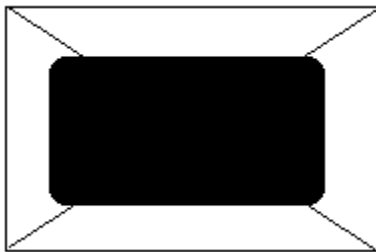
```

HTML file

```

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

```

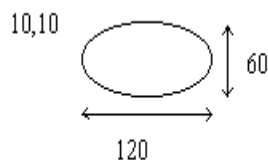


#### 4.2.6.2 Circles and ellipses

The `drawOval()` method can be used to draw a circle or an ellipse. It takes four arguments. The first two represent the top left corner of the imaginary rectangle and the other two represent the width and height of the oval.

If the width and height are equal, the oval becomes a circle.

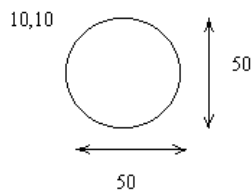
Eg;- `g.drawOval(10,10,120,60);`



The `fillOval()` method draws a solid Oval

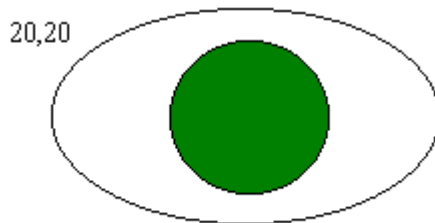
Eg;- `g.fillOval(10,10,50,50);`





Eg:-

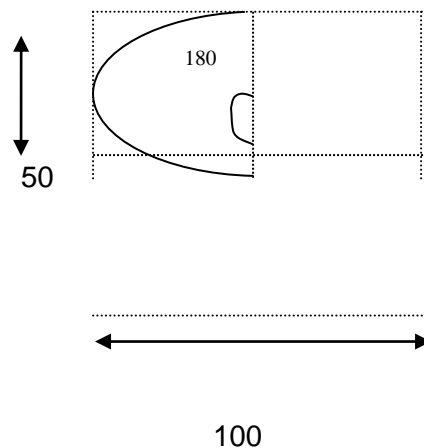
```
public void paint (Graphics g)
{
    g.drawOval(20,20,200,120);
    g.setColor(color.green );
    g.fillOval(70,30,100,100);
}
```



#### 4.2.6.3 Drawing arcs

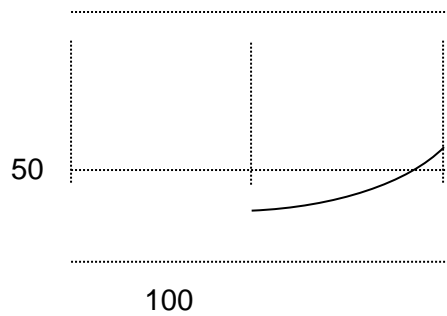
The drawArc() method is used to draw arcs. It takes six arguments. The first four are same as that of the drawOval( ) method and the last two represent the starting angle of the arc and the no. of degrees around the arc(sweep angle).

Eg:- g.drawArc(10,10,100,50,90,180)



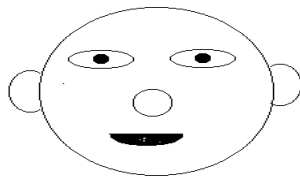
If the sweep argument is negative, then the arc is drawn in backward direction.

Eg:- g.drawArc(10,10,100,50,45, -135)



The fillArc( ) method is used to draw a filled arc. Filled arcs are drawn as if they were sections of a pie. Instead of joining the two end points, they are joined to the centre of the oval.

```
import java.awt.*;
import java.applet.*;
public class Face extends Applet
{
    public void paint (Graphics g)
    {
        g.drawOval (40,40,120,150); // Head
        g.drawOval (57,75,30,20); // Left eye
        g.drawOval (110,75,30,20); // Right eye
        g.fillOval (68,81,10,10); // Left pupil
        g.fillOval (121,81,10,10); // right pupil
        g.drawOval (85,100,30,30); // nose
        g.fillArc (60,12,80,40,180,180); //Mouth
        g.drawOval (25,92,15,30); //left ear
        g.drawOval (160,92,15,30) // right ear
    }
}
```



#### 4.2.6.4 Drawing

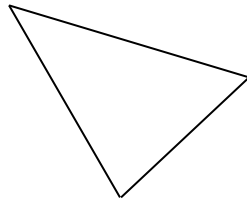
many sides. A polygon may be considered as a set of lines connected together. We can draw a polygon with n sides using the drawLine() method n times in succession.

```
Eg;- public void paint (Graphics g)
{
    g.drawLine (10,20,170,40);
    g.drawLine (170,40,80,140);
    g.drawLine (80,140,10,20);
}
```

#### polygons

Polygons are shapes with many sides. A polygon may be considered as a set of lines connected together.

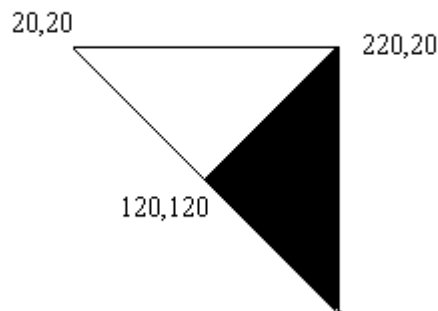
10,20



We can also draw using the `drawPolygon()` method. It contains 3 arguments. The first one is an array of integers having x coordinates. The second one is an array of integers having y coordinates and the third one is an integer for the total no. of points. Repeat the first point at the end for closing the polygon. The arrays x and y should be of same size.

```
public void paint (Graphics g)
{
    int x points [ ] = {10,170,80,10};
    int y points [ ] = {20,40,140,20};
    int n points = x points. Length;
    g.drawPolygon(x points, y points, n points);
}
```

A filled polygon can be drawn by calling the `fillPolygon()` method.



```
import java.awt.*;
import java.applet.*;
public class Poly extends Applet
{
    int x1[ ] = {20,120,220,20};
    int y1[ ] = {20,120,20,20};
    int n1 = 4;
    int x2[ ] = {120,220,220,120};
    int y2 [ ] = {120,20,220,120};
    int n2 = 4;
    public void paint (Graphics g)
    {
        g. draw Polygon (x1, y1,n1);
        g. fill Polygon (x2, y2, n2)
    }
}
```

We can also treat the polygon as an object

1. Define the x coordinate values as an array
2. Define the y coordinate values as an array
3. Define the no. of points n
4. Create a polygon object and initialize it with above values.
5. Call the drawPolygon() or fillPolygon() method with the polygon object as argument.

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

```
{
    int x[ ] = {20,120,220,20};
    int y[ ] = {20,120,20,20};
    int n =x.length;
    Polygon poly = new Polygon (x,y,n);
    g.drawPolygon(poly);
}
```

Now we can easily add points to the polygon using addPoint( ) using  
Object.addPoint(x,y);

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

```
{
    Polygon poly = new Polygon ( ); // creates empty polygon
    poly.addPoint(20,20);
    poly.addPoint(120,120);
    poly.addPoint (220,20);
    poly.addPoint(20,20);
    g.drawPolygon (poly);
}
```

#### 4.2.6.5 Drawing LineGraphs

A line graph is useful for displaying data or information that changes continuously over time. Another name for a line graph is a line chart. A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by straight line segments.

#### Applet to display Line Chart

```
import java.awt.*;
import java.applet.*;
public class Linechart extends Applet
{
    public void paint(Graphics g)
    {
        int x1[ ] = {20,120,200,225,250};
        int y1[ ] = {20,120,20,150,25};
        int n =x1.length;
        g.drawString("Y", 10, 20);
        g.drawString("X",300,200);
        g.drawLine(10,10,10,200);
        g.drawLine(300,200,10,200);
        for(int i=0;i<n;i++)
            g.drawLine(x1[i],y1[i],x1[i+1],y1[i+1]);
    }
}
```

**HTML file**

<HTML>

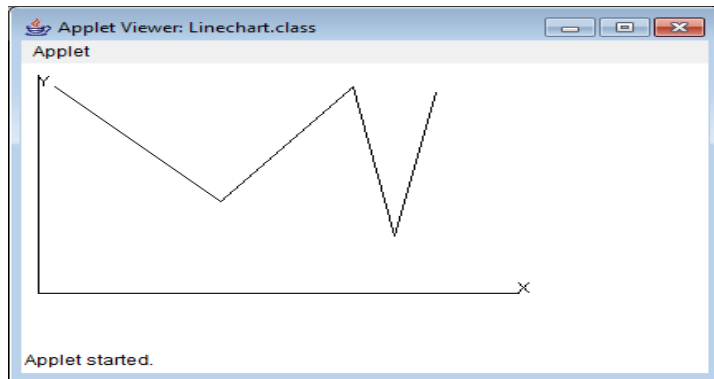
```

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

```

</HTML>

### Output



#### 4.2.6.6 Drawing Barchart

Applets can be designed to display bar charts, which are commonly used in comparative analysis of data.

Year	1991	1992	1993	1994
Turn Over Rs.crores	110	150	100	170

These data are input using the method `getParameter()`. The method `getParameter()` returns only string values and therefore the wrapper class method `parseInt()` is used to convert strings to the integer values.

#### Applet for drawing Bar Chart

```

import java.awt.*;
import java.applet.*;
public class Barchart extends Applet
{
    int n = 0;
    String label[];
    int value[];
    public void init()
    {
        n = Integer.parseInt(getParameter("columns"));
        label = new String[n];
    }
}

```

```

        value = new int[n];
        label[0] = getParameter("label1");
        label[1] = getParameter("label2");
        label[2] = getParameter("label3 ");
        label[3] = getParameter("label4");
        value[0] = Integer.parseInt(getParameter("c1"));
        value[1] = Integer.parseInt(getParameter("c2"));
        value[2] = Integer.parseInt(getParameter("c3"));
        value[3] = Integer.parseInt(getParameter("c4"));
    }
    public void paint(Graphics g)
    {
        for(int i =0;i<n;i++)
        {
            g.Setcolor(color.red);
            g.drawString(label[i],20,i*50+30);
            g.fillRect(50,i*50+10,value[i],40);
        }
    }
}

```

## HTML file

```

<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 = "label2" VALUE="92">
<PARAM NAME = "label3" VALUE="93">
<PARAM NAME = "label4" VALUE="94">
</HTML>

```

## 4.3 AWT Components and Event Handlers

### 4.3.1 Abstract Window Tool Kit

The Abstract Window Tool Kit (AWT) supports Graphical User Interface (GUI) programming. The AWT package contains set of classes for windows, buttons, labels, menus etc.

### 4.3.2 Events

Java is an event driven language. An event is an action triggered by the user such as a key press or mouse click. Java is an OOP language, so each event is considered as an object of class. Java provides a class called EventObject. The java.awt package contains a class called AWT event which is a subclass of EventObject.

The package java.awt contains several event classes.

### Event classes

1. KeyEvent – Generated when a key is pressed.
2. MouseEvent – Generated when mouse is operated.
3. WindowEvent – Generated when window is opened, closed etc.
4. ActionEvent.
5. AdjustmentEvent.
6. ComponentEvent.
7. FocusEvent.
8. InputEvent.
9. ItemEvent.
10. Text Event.

#### 4.3.2.1 Event Generators

An event is generated by an event source.

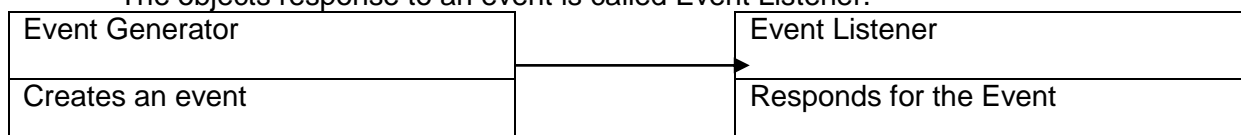
### Example

When a button is pressed, the ActionEvent is generated. Hence button is an event source.

Event	Generated By
Action Event	Button pressing Double clicking of a item in a list Selecting a menu item.
Item Event	Selection or deselection of a checkbox Change of choice item selection
Adjustment Event	Selection or deselection of checkable menu item.
Text Event	Manipulation of scrollbar.
Window Event	Enter a character in Text Field or Text Area. Open, close, iconify, deiconify, activate and deactivate windows.

#### 4.3.2.2 Event Listener

The objects response to an event is called Event Listener.

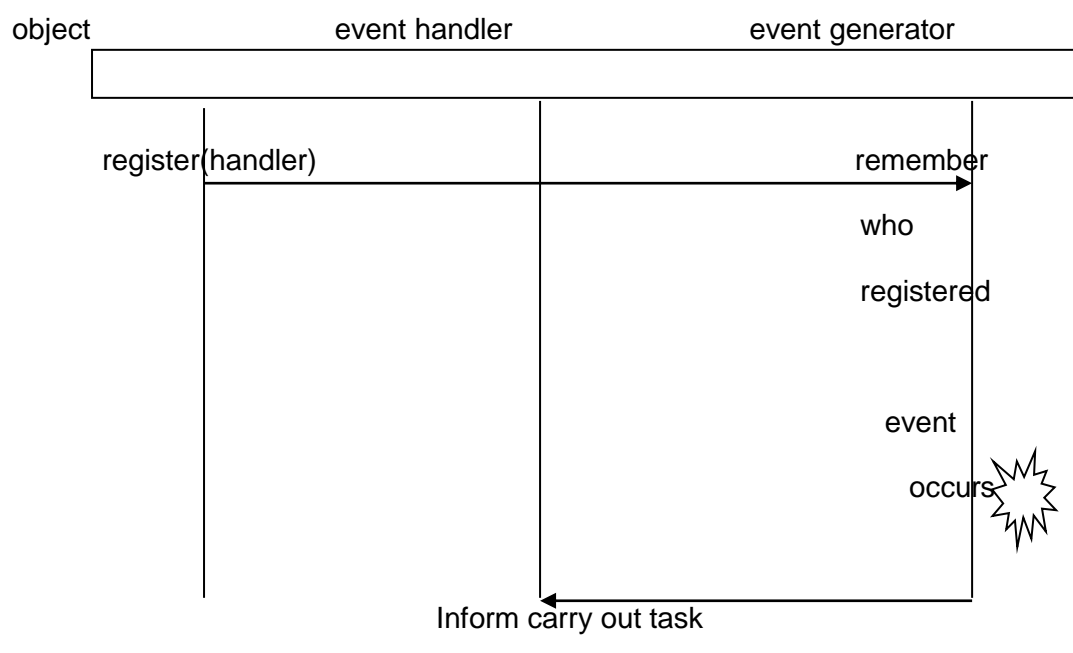


## Event Listeners

- ActionListener.
- AdjustmentListener.
- ComponentListener.
- ContainerListener.
- FocusListener.
- ItemListener.
- KeyListener.
- MouseListener.
- MouseMotionListener.
- WindowListener.

### 4.3.2.3 Event handling

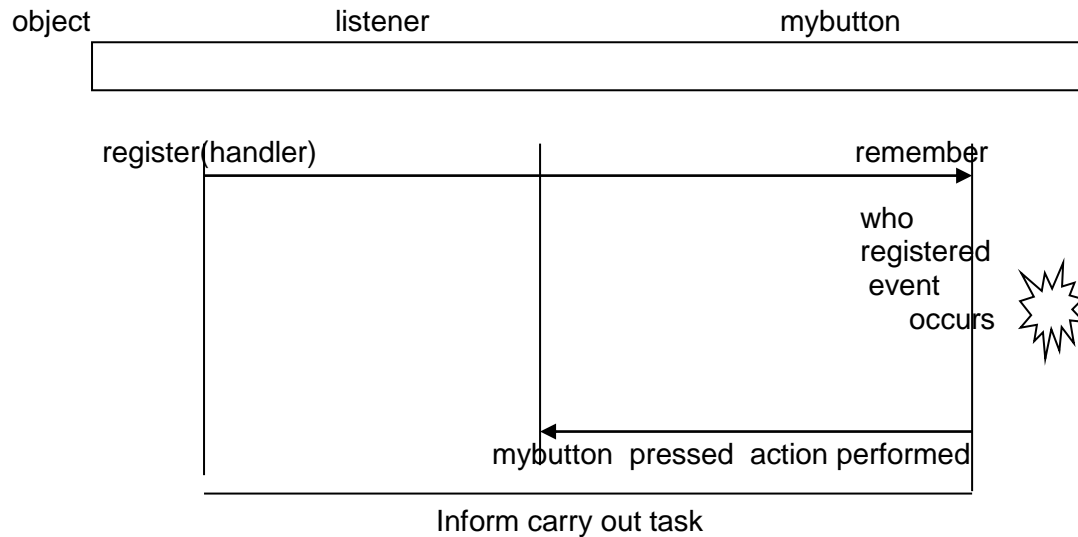
The programmer developing an applet or application will write event handlers. The programmer will also need to register the handler with the appropriate event generator.





Buttons generate Action Events. An event handler is technically called a listener in Java. A listener is registered with a button, by some object sending it the addActionListener method, with a parameter indicating who the listener ( handler ) is going to be. The button remembers who is registered. This is done automatically by code in the Java libraries.

When the event occurs, the button informs the listeners that the event occurred by sending the action performed message to each of the listeners. This is also done automatically by Java library code.



#### 4.3.3.1 Labels

The label component is the simplest of Java's AWT and consists of a text string for display only.

##### Syntax

```
Label labelname = new label ( " Text message " );
```

##### Example

```
Label L1 = new label ( "Comp Engg " );
```

The above example creates a label with name L1 and displays the message

Comp Engg.

##### Methods

Method Name	Purpose	Example
setText ( )	To change the text of a label	L1.setText ( " Mech . Engg " );
getText ( )	To get the text in the label to a string variable	String S1 = L.getText ( );
setAlignment ( )	To change the alignment of the label	L1.setAlignment ( label . LEFT )  Like this we can align Left, Right , Center

getAlignment ( )	To get the alignment of the label.	int x = L1.getAlignment ( );
------------------	------------------------------------	------------------------------

### Example

```
import java . awt . * ;
import java . applet . * ;
public class Labeldemo extends Applet
{
    public void init ( )
    {
        Label one = new Label ( "One" );
        Label two = new Label ( "Two" );
        Label three = new Label ( "Three" );

        add ( one );
        add ( two );
        add ( three );
    }
}
```

add ( one ) of method is used to add the label one to the applet..

### HTML file

```
< HTML >
< BODY >
< APPLET

                                CODE = Labe demo . class
                                WIDTH = 400
                                HEIGHT = 400 >

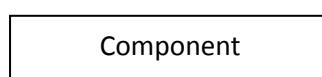
< / APPLET >
< / BODY >
< / HTML >
```

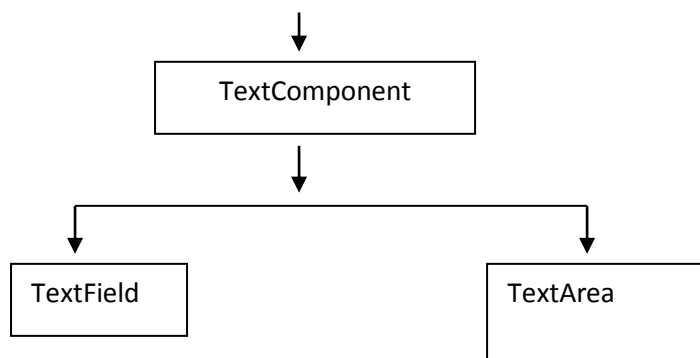
### Output

Applet Viewer : Label demo
Applet

### 4.3.3.2 Text -Component

The TextField and TextArea classes implement one dimensional and two dimensional components for text input display and editing. Both classes extend from the TextComponent class.





#### 4.3.3.2.1 TextField

It is used for Single line text entry. It is also called as edit control.

##### Constructors

1. `TextField ( )` → Constructs an empty text field.
2. `TextField ( int cols )` → Constructs an empty text field with specified number of columns.
3. `TextField ( String text )` → Constructs a textfield whose initial content is text.
4. `TextField ( String text , int cols )` → Constructs a textfield whose initial content is text with the specified number of columns.

##### Methods

- `getSelectedText ( )` → Returns the currently selected text.
- `getText ( )` → Returns the text content of the component.
- `setText ( )` → Sets the text content of the component.

##### Example

`TextField tf1 = new TextField ( " Java" );` - It creates a text field with the content Java.

`add( tf1 )` → Add the textfield to the applet.

#### 4.3.3.2.2 TextArea

This is used for **Multiline text entry**.

##### Constructors

1. `TextArea( )` → Constructs an empty text area.
2. `TextArea ( int rows , int cols )` → Constructs an empty text area with the specified no. of rows and columns.
3. `TextArea ( String text )` → Constructs a text area whose initial content is text.
4. `TextArea ( String text , int rows, int cols )` → Constructs a text area whose initial content is text with the specified number of rows and columns

##### Methods

- `getSelectedText ( )` → Return the currently selected text.

- `getText ( )` → Returns the text content of the component.
- `SetText ( )` → Sets the text content of the component.

### Example

```
TextArea ta = new TextArea ( 4, 3 );
    Creates a text area with 4 rows and 3 columns.
add ( ta ) → Add the textArea to the applet.
```

### 4.3.3.3 Buttons

A push button is a component that contains a label and generates an event when it is pressed.

#### Constructors

```
Button ( ) → Creates an empty button.
Button ( String Str ) → Creates a button that contains Str as a label.
```

#### Syntax

```
Button b1 = new Button ( "Text" );
```

### Example

```
Button b1 = new Button ( "cancel" );
This creates a button with label "cancel" .
```

#### Methods

1. `add ( )` → To add a button to the applet.
2. `setLocation (int x , int y )` → ( x, y ) denotes the co-ordinates of the top left corner of the button.
3. `setSize ( int w, int h )` → ( w, h ) denotes the width and height of the button.
4. `setBounds ( int x, int y, int w, int h )` → ( x, y ) ⇒ coordinates of the top left corner.

( w, h ) ⇒ width and height of button

in pixels.

### Example

```
import java . applet . * ;
import java . awt . * ;
public class buttondemo extends Applet
{
    public void init ( )
    {
        Button b1 = new Button ( "Mech" );
        Button b2 = new Button ( "Comp" );
```

```

        add ( b1 );
        add ( b2 );
    }
}

```

#### 4.3.3.4 Check boxes

A checkbox is a control that is used to turn an option on or off. It consists of a small box that can either contain a checkmark or not. There is a label associated with each checkbox. The checkbox has two states, true ( checked ) and false ( not checked ). The check boxes can be used individually or as part of a group.

##### Constructors

- Checkbox ( )
- Checkbox ( String S )
- Checkbox ( String S, boolean state )

If the state is not specified the default state is false.

##### Methods

1. setLabel ( ) → used to assign label for the checkbox.
2. getLabel ( ) → used to get the label of the checkbox.
3. setState ( ) → used to assign state for the checkbox.
4. getState ( ) → Returns a boolean value representing the state of checkbox.

##### Example

```

import java . applet . * ;
import java . awt . * ;
public class checkbox extends Applet
{
    public void init ( )
    {
        Checkbox C1 = new checkbox ( "Red" );
        Checkbox C2 = new checkbox ( "green" , true );
        add ( C1 );
        add ( C2 );
    }
}

```

#### 4.3.3.5 Choice

The choice class is used to create a pop-up list of items from which the user may choose. A choice control is a form of menu. Each item can be added using addItem( ) method.

##### Syntax

```

Choice C1 = new choice( );

```

New items can be added to the menu using addItem( ) method.

##### Example

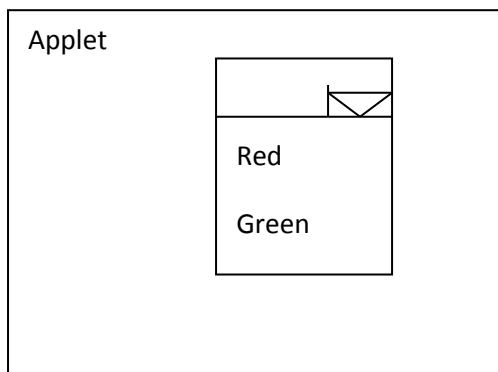
C1. addItem( "Item" );

### Methods

1. addItem ( ) → Adds new items to the menu.
2. countItem ( ) → To count number of items in the choice menu.
3. getItem ( ) → To get an item of the choice menu.
4. getSelectedItem ( ) → To get the item which has been selected.

### Example

```
import java . applet . * ;
import java . awt . * ;
public class choices extends Applet
{
    choice opt = new Choice ( );
    public void init ( )
    {
        opt.addItem ( "Red" );
        opt.addItem ( "Green" );
        opt.addItem ( "White" );
        opt.addItem ( "Yellow" );
        add ( opt );
    }
}
```



### 4.3.3.6 Scroll bars

The scrollbar class provides a user interface to scroll through a range of integer values.

### Constructors

1. Scrollbar ( ) → Constructs a vertical scroll bar.
2. Scrollbar ( int orientation ) → Constructs a scrollbar with the specified orientation. (Scrollbar.HORIZONTAL, Scrollbar.VERTICAL).
3. Scrollbar ( int orient , int value , int visible, int min, int max )

⇒ Constructs a scroll bar with the specified orientation, value of the scroll bar, size of the visible portion, minimum and maximum value.

### Example

```
Scrollbar mark;  
mark = new Scrollbar ( Scrollbar.HORIZONTAL, 42, 10, 0, 100 );
```

This constructs a horizontal scrollbar with  
value = 72

visible Size = 10  
minimum value = 0  
maximum value = 100

### Methods

1. `getValue ( )` → Returns the current value in scrollbar.
2. `setValue( int v )` → Sets the value v to the scrollbar.
3. `getMinimum( )` → Returns the minimum value.
4. `getMaximum( )` → Returns the maximum value.
5. `getVisible( )` → Returns the visible size.
6. `getOrientation( )` → Returns the integer value representing the orientation.

#### 4.3.3.7 MenuBar and Menuclass

A menu bar displays a list of Top-level menu choices. The menu bar has a horizontal bar in which the menu title and the pull down menu appears on the Screen.

#### Classes used

1. MenuComponent.
2. MenuItem.
3. MenuBar.
4. Menu.
5. Checkbox MenuItem.

##### 4.3.3.7.1 Menu Component Class

The Menu Component class is an abstract class from which the MenuBar and MenuItem classes are derived.

#### Methods

1. `setFont ( Font f )` → Sets the display font for the Menu Component object.
2. `getFont( )` → Gets the font of the Menu Component and returns the font object.

##### 4.3.3.7.2 Menu Bar Class

The MenuBar class is derived from MenuComponent Class.

#### Syntax

```
MenuBar mb = new MenuBar( )
```

creates an empty Menu Bar. A menu object can be added using add( ) method. Suppose mb is a MenuBar object and rep is a MenuObject the command  
mb.add( rep )

#### 4.3.3.7.3 Menu Item class

The MenuItem object have a String as argument.

##### Syntax

```
MenuItem mi = new MenuItem ( String S );
```

##### Methods

1. getLabel( ) → Returns the label string of the MenuItem.
2. setLabel( String S ) → Sets a new label for the MenuItem.
3. isEnabled( ) → Returns a Boolean value representing whether the MenuItem is enabled.
4. enable( ) → Enables MenuItem.

#### 4.3.3.7.4 Menu Class

Menu is a derived class of MenuItem class. This means that a Menu object can be considered as a MenuItem.

##### Syntax

```
Menu mu = new Menu ( String S, boolean b )
```

This method constructs a Menu object with label s. When the boolean value b is true this creates a tear-off menu. A tear-off menu is the one which is still displayed after click and release the mouse button.

##### Method

1. isTearOff ( ) → This Boolean method returns whether the menu is a tear-off menu.
2. countItems ( ) → Returns the number of items in the menu.
3. getItem( int i ) → Returns the MenuItem object at the i<sup>th</sup> index of the menu.
4. add ( MenuItem m ) → Adds the MenuItem m to the menu.

##### Example

```
import java.applet.*;
import java.awt.*;
public class samplemenu extends Frame
{
    MenuBar mainmenu;
    Menu file;
```



```

MenuItem op,ne,sa,qu;
Menu help;
MenuItem commands,about;
Sample menu()
{
    setTitle ("Menu Demo");
    setLayout (new FlowLayout());
    mainmenu = new MenuBar();
    setMenuBar (mainmenu);

    file = new Menu("File");
    op = new MenuItem("Open");
    ne = new MenuItem("New");
    qu = new MenuItem("Quit");
    sa = new MenuItem("Save");
    file.add(op);
    file.add(ne);
    file.add(sa);
    file.add(qu);

    mainmenu.add(file);
    help = new Menu("Help");
    commands = new MenuItem("commands");
    about = new MenuItem("About");

    help.add(commands);
    help.add(about);
    mainmenu.add(help);
}
public static void main(String args[])
{
    Frame f = new samplemenu();
    f.setBounds(1,1,400,400);
    f.setVisible(true);
}
}

```

### Output

Menu Demo	
File	Help
open new save Quit	

#### 4.3.4 Layout Managers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers.

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout

4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

#### 4.3.4.1 BorderLayout

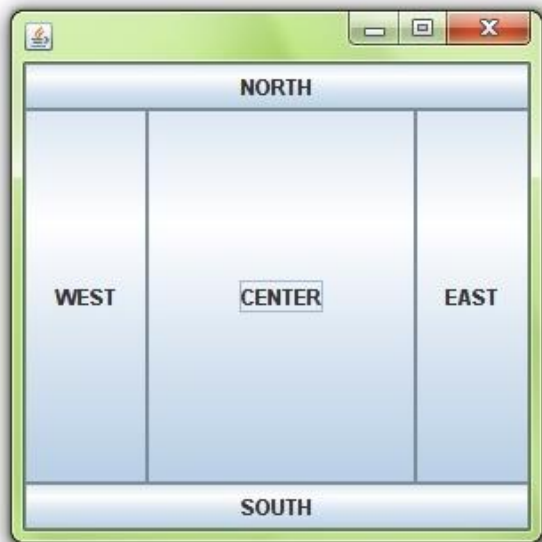
The BorderLayout is used to arrange the components in five regions: **NORTH, SOUTH, EAST, WEST AND CENTER**. Each region may contain one component only.

1. public static final int NORTH
2. public static final int SOUTH
3. public static final int EAST
4. public static final int WEST
5. public static final int CENTER

#### Constructors

- BorderLayout() - creates a border layout but with no gaps between the components.
- JBorderLayout(int hgap, int vgap) - creates a border layout with horizontal and vertical gaps between the components.

#### Example



```
import java.awt.*;
import javax.swing.*;
public class Border
{
    JFrame f;
    Border()
    {
        f=new JFrame();

        JButton b1=new JButton("NORTH");;
        JButton b2=new JButton("SOUTH");;
        JButton b3=new JButton("EAST");;
        JButton b4=new JButton("WEST");;
        JButton b5=new JButton("CENTER");;
```

```

        f.add(b1, BorderLayout.NORTH);
        f.add(b2, BorderLayout.SOUTH);
        f.add(b3, BorderLayout.EAST);
        f.add(b4, BorderLayout.WEST);
        f.add(b5, BorderLayout.CENTER);
        f.setSize(300,300);
        f.setVisible(true);
    }
}

public static void main(String[] args)
{
    new Border();
}

```

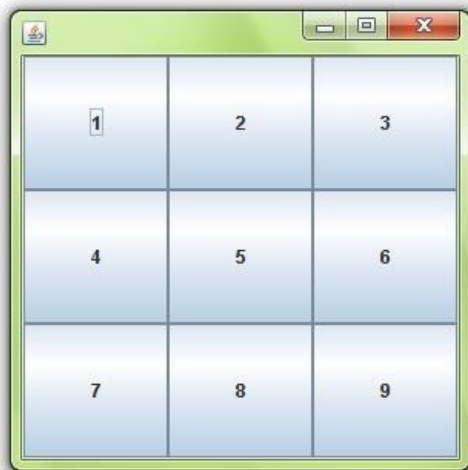
#### 4.3.4.2 GridLayout

The GridLayout is used to arrange the components in rectangular grid.

##### Constructors

- GridLayout() - creates a grid layout with one column per component in a row.
- GridLayout(int rows, int columns) - creates a grid layout with the given rows and columns but no gaps between the components.
- GridLayout(int rows, int columns, int hgap, int vgap) - creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

##### Example



```

import java.awt.*;
import javax.swing.*;

public class MyGridLayout
{
    JFrame f;
    MyGridLayout()
    {
        f=new JFrame();

        JButton b1=new JButton("1");
        JButton b2=new JButton("2");

```

```

        JButton b3=new JButton("3");
        JButton b4=new JButton("4");
        JButton b5=new JButton("5");
        JButton b6=new JButton("6");
        JButton b7=new JButton("7");
        JButton b8=new JButton("8");
        JButton b9=new JButton("9");
        f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
        f.add(b6);f.add(b7);f.add(b8);f.add(b9);

        f.setLayout(new GridLayout(3,3));

        f.setSize(300,300);
        f.setVisible(true);
    }
}
public static void main(String[] args)
{
    new MyGridLayout();
}

```

#### 4.3.4.3 FlowLayout

The FlowLayout is used to arrange the components in a line, one after another. It is the default layout of applet or panel.

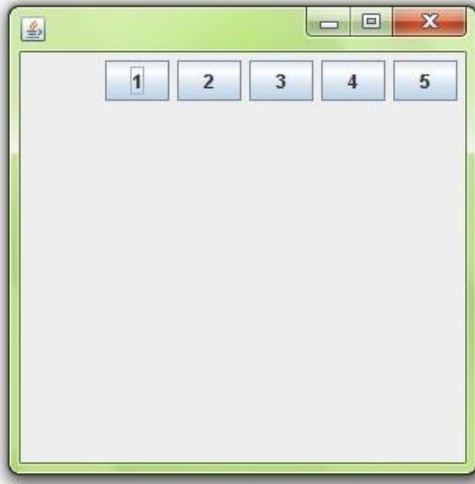
##### Fields

1. public static final int LEFT
2. public static final int RIGHT
3. public static final int CENTER
4. public static final int LEADING
5. public static final int TRAILING

##### Constructors

- FlowLayout() - creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
- FlowLayout(int align) - creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
- FlowLayout(int align, int hgap, int vgap) - creates a flow layout with the given alignment and the given horizontal and vertical gap.

##### Example



```
import java.awt.*;
import javax.swing.*;

public class MyFlowLayout
{
    JFrame f;
    MyFlowLayout()
    {
        f=new JFrame();
        JButton b1=new JButton("1");
        JButton b2=new JButton("2");
        JButton b3=new JButton("3");
        JButton b4=new JButton("4");
        JButton b5=new JButton("5");
        f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
        f.setLayout(new FlowLayout(FlowLayout.RIGHT));
        f.setSize(300,300);
        f.setVisible(true);
    }
}

public static void main(String[] args)
{
    new MyFlowLayout();
}
```

## 4.3.5 Input Events

### 4.3.5.1 Key events

When an event is generated through the key board, the KeyHandler class responds. We can generate three types of key events. They are

1. Pressing a key (holding a key pressed)

2. Releasing a pressed key
3. Typing a key

The KeyListener interface has three methods corresponding to the above events. They are

```
keyPressed (KeyEvent e)
keyReleased (KeyEvent e)
keyTyped (KeyEvent e)
```

All the methods take an object of KeyEvent class as the parameter. The KeyEvent class has a method getKeyChar ( ) which returns the character of the key from which the event is generated.

Eg:-

```
if e is a KeyEvent, then
    Char C;
    C = e.getKeyChar ( );
```

Returns the character in C.

Eg:-

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class KeyEventDemo extends Applet
{
    String msg = "";
    public void init ( )
    {
        addKeyListener (new KeyManager ( ));
        requestFocus ( );
    }
    public void paint (Graphics g)
    {
        font f = new Font ("Arial", Font Bold, 30);
        g.setFont (f);
        g.drawString (msg, 50, 50);
    }
}
class KeyHandler implements KeyListener
{
    public void keyPressed (KeyEvent e)
    {
        char c = e.getKeyChar ( );
        switch ( )
        {
            case 'R':
            case 'r' : setBackground (new Color (255,0,0));
                        msg = "Red";
                        break;
            case 'G':
            case 'g' : setBackground (new Color (0,255,0));
                        msg = "green";
                        break;
            case 'B':
```

```

        case 'b': setBackground (new Color (0,0,255));
                    msg = "blue"
                    break;
            }
        repaint ( );
    }

    public void keyReleased (KeyEvent e)
    {
    }
    public void keyTyped (KeyEvent e)
    {
    }
}
}

```

#### 4.3.5.2 Mouse Events

The MouseEvent class is used to handle the events generated by the mouse. There are two types of mouse events. They are

1. Events associated with the click of button or position of mouse pointer.
2. Events associated with the motion of mouse.

##### Mouse click

The events associated with click of button or position of mouse pointer are listed below.

- mousePressed ( )
- mouseReleased ( )
- mouseClicked ( )
- mouseEntered ( )
- mouseExited ( )

All the events use the MouseEvent object as parameter. The methods must implement the MouseListener Interface.

##### Mouse Motion

There are two events associated with the movement of mouse. They are

- mouseMoved ( )
- mouseDragged ( )

These methods must implement the MouseMotionListener interface

```

import java. awt. *;
import java. applet.*;
import java. awt. event. *;
public class MouseDemo Extends Applet
{
    String msg = " ";
    public void init ( )

```

```

        {
            add MouseListener (new MouseManager ( ));
            add MouseMotionListener(new MouseMotioner ( ));
        }
    public void paint (Graphics g)
    {
        Font f1 = new Font ("Ariel", Font.BOLD, 20);
        g.setFont (f1);
        g.drawString (msg, 100, 50);
    }
}
Class MouseManager implements Mouse Listener
{
    public void mousePressed (MouseEvent e)
    {
        msg = "Mouse Pressed";
        repaint ( );
    }
    public void mouseReleased (MouseEvent e)
    {
        msg = "Mouse Released";
        repaint ( );
    }
    public void mouseClicked (MouseEvent e)
    {
        msg = "Mouse clicked",
        repaint ( );
    }
    public void mouseEntered (MouseEvent e)
    {
        msg = "Mouse Entered";
        repaint ( );
    }
    public void mouseExited (MouseEvent e)
    {
        msg = "Mouse Exited";
        repaint ( );
    }
}
class MouseMotioner implements MouseMotionListener
{
    public void MouseMoved (MouseEvent e)
    {
        msg = "Mouse Moved";
        repaint( );
    }
    public void MouseDragged (MouseEvent e)
    {
        msg = "Mouse Dragged";
        repaint ( );
    }
}
}

```



## Review Questions

### Part A (2 marks)

1. What are packages? What are its types?
2. What are applets?
3. How will you draw a polygon?
4. What is an event?
5. List the methods of Label class.
6. What are event generators?
7. What are layout managers?

### Part B (3 marks)

1. List the different system packages?
2. What are the types of applets?
3. List differences between applets and application programs?
4. How will you pass parameters to applets?
5. How will you draw lines and rectangles?
6. What are the types of events?
7. What are the different mouse events?

### Part C (5 marks)

8. How will you create a package?
9. Explain the life cycle of an applet with example.
10. List the various Applet class methods.
11. Give the syntax of the Applet tag.
12. Explain any three methods of Graphics class with example.
13. How will you draw a bar chart?
14. Explain how will you create a menu with example.
15. Explain any one Layout Manager with example.
16. Explain key events with example?

## **UNIT V - EXCEPTION HANDLING, MULTITHREADS AND I/O STREAMS**

### **OBJECTIVES**

- To know different types of errors.
- To understand the reasons & advantages of exception
- To understand how to handle exception
- To know what is multithreading
- To know the lifecycle of a thread
- To know the different thread methods
- To know how to define and run a thread
- To understand what is thread priority, thread synchronization and thread scheduling
- To know what is a file
- To know what is stream and its types
- To know the different stream classes
- To understand the methods used in file operations

### **5.1 Exception Handling**

#### **Exception**

An exception is a problem that arises during the execution of a program. When an exception occurs, the normal flow of the program is disrupted and the program terminates abnormally, therefore these exceptions are to be handled. An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- Try to divide a number by zero
- A file that needs to be opened cannot be found.

#### **5.1.2 Advantages of Exception Handling**

1. It allows us to control the normal flow of the program and avoids abnormal situation.
2. It also gives us the scope of organizing and differentiating between different error types using a separate block of codes.

#### **5.1.3 Types of Errors**

##### **Three types of errors**

There are basically three types of errors

1. Syntax errors
2. Runtime errors
3. Logic errors

##### **Syntax errors**

The syntax errors represent grammar errors in the use of the programming language.

Common examples are:

- Misspelled variable and function names
- Missing semicolons
- Improperly matched parentheses, square brackets, and curly braces
- Use of undeclared variables

##### **Runtime errors**

Runtime errors occur when a program with no syntax errors asks the computer to do something that the computer is unable to reliably do. Common examples are:

- Trying to divide by a value by zero
- Trying to open a file that doesn't exist
- Trying to access an element that is out of the bounds of an array
- Converting invalid string to number

##### **Logic errors**

Logic errors occur when there is a design flaw in the program. Common examples are:

- Multiplying instead of dividing
- Opening and using data from the wrong file

- Displaying the wrong message

#### 5.1.4 Basics of Exception Handling

Exception Handling is a mechanism used to handle the exceptions. Java provides number of classes to handle exceptions. It uses try-catch block to handle the exception.

```
try
{
    Statements that causes an exception
}
catch(Exception class 1 object1)
{
    Statements that handles the exception
}
-----
catch(Exception class n objectn)
{
    Statements that handles the exception
}
finally
{
    Statements }
```

##### try block

It check the statements for exception. If exception occurs, it throws it.

##### catch block

The catch block follows the try block .It catches the exception thrown by the try block and checks with the exception type. If match occurs, the statements inside the block will be executed. A program may have multiple catch blocks.

##### finally block

It is an optional block. If it is present, it should after all the catch blocks. This block is executed, If exception is thrown or not.

Example

```
public class Example6
{
    public static void main(String args[ ])
    {
        int a=30,b=5,c=5,d;
        try
        {
            d = a/(b-c);
            System.out.println("d= "+d);
        }
        catch (ArithmeticException e)
        {
            System.out.println("Divide by zero ");
        }
        finally
        {
            System.out.println("Exception over");
        }
    }
}
```

### 5.1.5 try block

It check the statements for exception. If exception occurs, it throws it.

```
try
{
    Statements that causes an exception
}
```

Eg:

```
try
{
    Int a[ ]= {10,20,30};
    System.out.println(a[3]);
}
```

### 5.1.6 throwing an exception

Normally the try block throws the exception. Suppose if we want to throw the exception, the throw keyword is used.

throw new exception class;

```
eg:    throw NumberFormatException();
public class Example9
{
    public static void main(String args[ ])
    {
        int a=30,b=15,c=5,d;
        try
        {
            if ((b-c) ==0) throw new ArithmeticException();
            d = a/(b-c);
            System.out.println("d= "+d);
        }
        catch (ArithmeticException e)
        {
            System.out.println("Divide by zero ");
        }
        finally
        {
            System.out.println("Exception over");
        }
    }
}
```

### 5.1.7 catching an exception

The catch block follows the try block .It catches the exception thrown by the try block and checks with the exception type. If match occurs, the statements inside the block will be executed. A program may have multiple catch blocks

```
catch(Exception class object)
{
    Statements to handle the exception
}
```

Eg:

```
catch(IOException e1)
{
    System.out.println(e1);
}
```

### 5.1.8 finally statement

It should be present after all the catch blocks. This block is executed, If exception is thrown or not.

```
finally
{
    Statements
}
```

Eg:

```
finally
{
    System.out.println("Exception over");
}
```

## 5.2 Multithreading

A thread is the path followed when executing a program. All Java programs have at least one thread known as the main thread. Every Java thread is created and controlled by the **java.lang.Thread** class.

Java supports multithreading. Multithreading is the process of executing multiple threads simultaneously.

### 5.2.1 Creating Threads

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### 5.2.2 Life of a thread

A thread can have five different states, during its life time a thread is always in any one of these states and can move from one state to another. Followings are the states of a thread:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

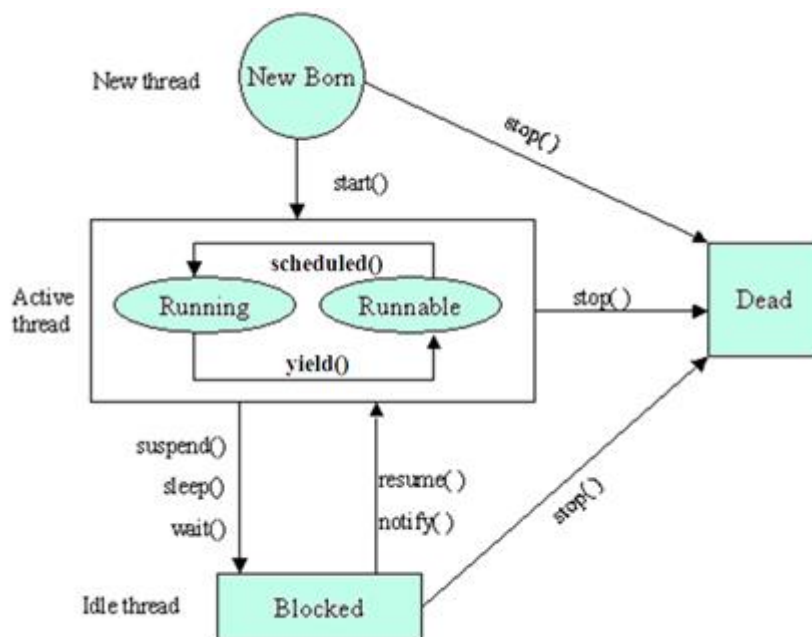


Fig 5.1 Life cycle of Thread

1. **Newbornstate:** A newborn state is a state when a thread object is created, and is not yet scheduled for running. It can move to runnable state using start() method or move to dead state using stop() method.
2. **Runnable state:** The runnable state means the thread is ready for the execution and waiting for processor.
3. **Running state:** In this state the processor has given the time to the thread for execution. From this state, thread enters into blocked (when suspend(),sleep(), or wait() is called ) or dead state (when stop() is called).
4. **Blocked state:** If we preventing the thread from entering into runnable/running state then this state called as blocked state. It goes to Runnable state when resume(),notify() is called or specified time expires.
5. **Dead state:** When a thread has completed its execution using run() method it is called the natural death. And when we stopped or moved a thread to dead state by calling stop() method it causes a premature death.

### 5.2.3 Defining & Running Thread

The thread can be defined by extending the Thread class.

1. Define subclasses by extending the Thread class

```
class classname extends Thread
{
}
```

2. Override the run() method in all subclasses.

```
public void run ( )
{
}
```

3. Create thread object in the main() method

```
classname object = new classname();
```

4. Call the start method using the thread object.

```
Object.start();
```

Example

```
class Even extends Thread
{
    public void run()
    {
        for(int i=0; i<=10; i=i+2)
            System.out.println("Even number: "+i);
    }
}
```

```

}
class Odd extends Thread
{
    public void run()
    {
        for(int j=1; j<=10; j=j+2)
            System.out.println("Odd number: "+j);
    }
}
public class Example7
{
    public static void main(String args[ ])
    {
        Even e1 = new Even();
        Odd o1 = new Odd();
        e1.start();
        o1.start();
    }
}

```

#### 5.2.4 Thread Methods

Method	Use
void run()	It is used to perform action for a thread
void start()	It starts the execution of the thread
void sleep(long miliseconds)	It stops the currently executing thread for the specified number of milliseconds.
void suspend()	It is used to suspend the execution of the thread.
void yield()	It is used to bring the thread to runnable state.
void stop()	It is used to stop the thread.
void resume()	It is used to resume the suspended thread.
void wait()	It stops the currently executing thread temporarily.
void notify()	It is used to bring the thread to runnable state.
boolean isAlive()	It tests if the thread is alive. If alive, it returns true, otherwise false.

#### 5.2.5 Thread Priority

Each thread have a priority. Priorities are represented by a number between 1 and 10. Default priority of a thread is 5 (NORM\_PRIORITY). Normally the OS assigns same priority to all threads. The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. Generally higher-priority threads get more CPU time than lower-priority threads. Two methods are used to set and the priorities of the thread.

##### **set\_priority() method**

To set a thread's priority, the **setPriority( )** method is used.

```
final void setPriority(int level)
```

The *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range MIN\_PRIORITY and MAX\_PRIORITY.

Eg: `t1.setPriority(7);`

#### **get\_priority() method**

To get the current priority of the thread, **getPriority( )** method is used.

```
final int getPriority( )
```

Eg: `t1.getPriority();`

### **5.2.6 Synchronization**

Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Synchronization is achieved by using the concept monitor(semaphore). Only one thread can own a monitor at a time. when a thread acquires a lock, it is said to be entered the monitor. All other threads trying to acquire that lock are suspended until the first thread exists the monitor. To achieve this, java uses synchronized keyword before the method. Synchronized method is used to lock an object for any shared resource.

```
synchronized void methodname()
{
}
}
```

Example

```
class Even extends Thread
{
    synchronized public void run()
    {
        for(int i=0; i<=10; i=i+2)
            System.out.println("Even number: "+i);
    }
}
```

### **5.2.7 Implementing Runnable Interface**

The thread can also be defined by implementing the runnable interface.

1. Define subclasses by implementing the Runnable interface

```
class classname implements Runnable
{
}
}
```

2. Override the run() method in all subclasses.

```
public void run ( )
{
}
}
```

3. Create thread object in the main() method

```
classname object = new classname();
```

4. Call the start method using the thread object.

```
new Thread(Object).start();
```

Example



```

class Even implements Runnable
{
    public void run()
    {
        for(int i=0; i<=10; i=i+2)
            System.out.println("Even number: "+i);
    }
}
class Odd implements Runnable
{
    public void run()
    {
        for(int j=1; j<=10; j=j+2)
            System.out.println("Odd number: "+j);
    }
}
public class Example8
{
    public static void main(String args[ ])
    {
        Even e1 = new Even();
        Odd o1 = new Odd();
        new Thread(e1).start();
        new Thread(o1).start();
    }
}

```

### 5.2.8 Thread Scheduling


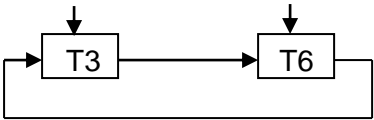

Scheduling is a process of allocating the CPU for execution of the threads. This allocation is done based on the priority of the threads. This process is done with the help of operating system.

The operating system selects the thread with highest priority and allocates the CPU for execution, other threads are waiting until it completes its execution. After completing, the OS gives chance to the next highest priority thread.

If two or more threads have the same priority, the OS allocates the CPU in round robin manner. This process is continued for all threads.

Eg:

Thread	T1	T2	T3	T4	T5	T6
Priority	7	9	8	2	4	8

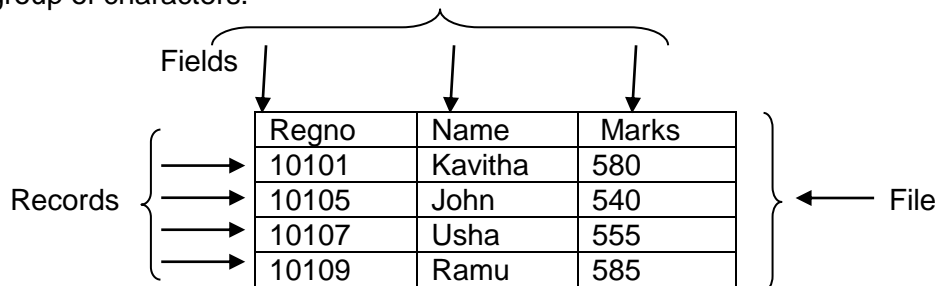
Priority	Thread name	Allocation of CPU
9	T2	
8	T3, T6	
7	T1	

4	T5	↓ T5
2	T4	↓ T4

### 5.3 I/O streams

#### 5.3.1 File

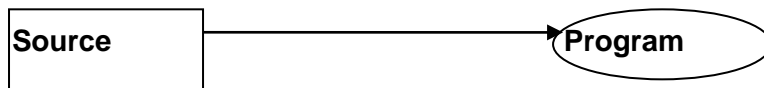
A file is a collection of related records. A record is the collection of fields. A field is a group of characters.



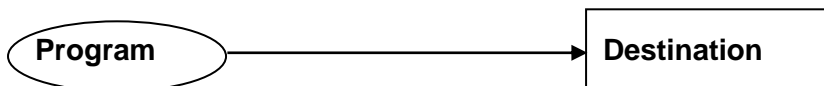
#### 5.3.2 Streams

A stream is the path along which data flows. There are two kinds of Streams:

1. **Inputstream** – The Inputstream reads data from a source and sends it to the program.



2. **Outputstream** – The Outputstream takes the data from the program and sends it to the destination.



#### 5.3.3 Advantages

1. Complex file processing operations are easily done.
2. It provides a clean abstraction for a complex task.
3. Serialization of I/O classes provides a solution to serialization of objects.
4. It is used to filter data along the pipeline of streams, so that we can obtain data in a desired format.

#### 5.3.4 The stream classes

The **java.io** package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

1. **Byte stream** ---- to perform input and output operations on bytes.

## 2. **Character stream** ---- to perform input and output operations on Characters

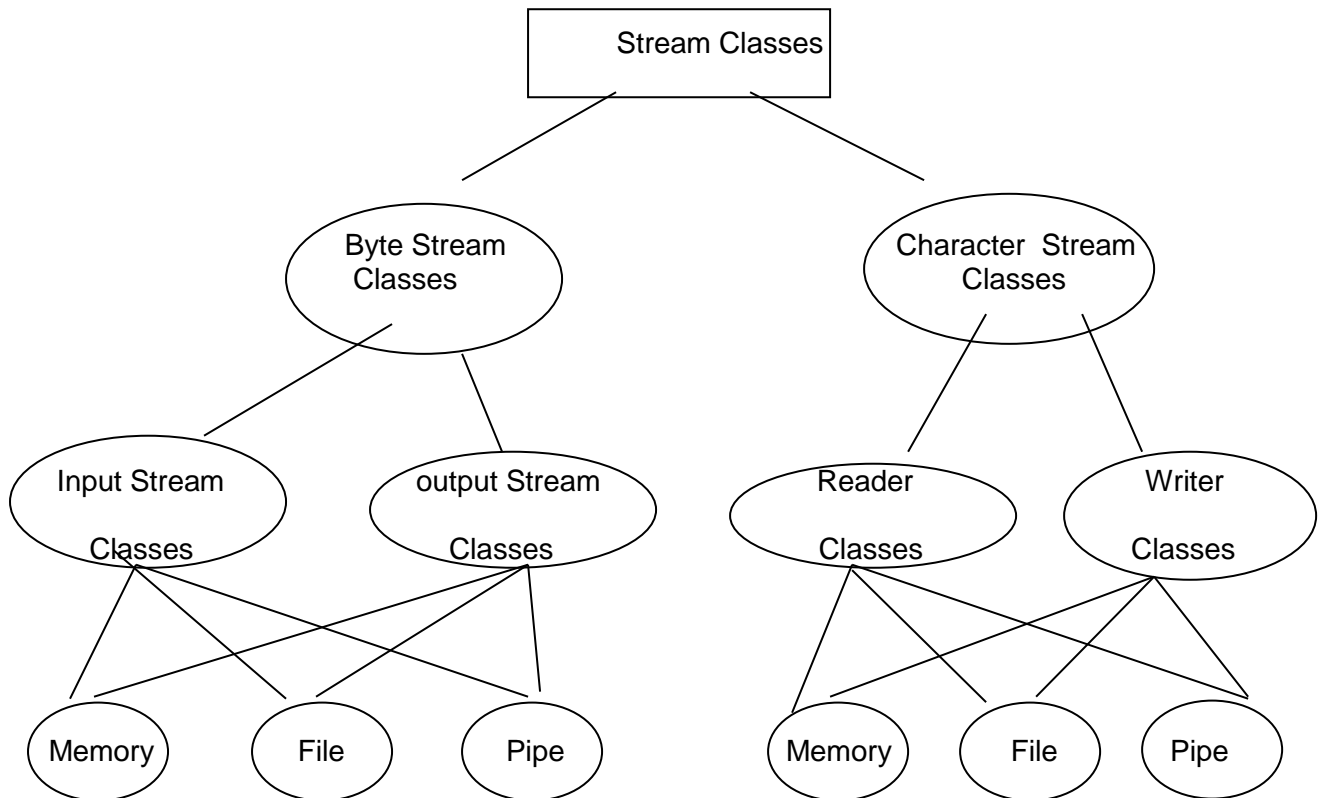


Fig 5.2 Classification of java stream classes

### 5.3.5 Byte streams

Java byte streams are used to perform input and output operations on bytes. Though there are many classes related to byte streams but the most frequently used classes are

**InputStream** and **OutputStream**.

#### **InputStream class**

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes. The methods in this class are given below:

Method	Use
int available()	It returns the number of bytes that can be read from the current input stream.
int read()	It reads the next byte of data from the input stream. It returns -1 at the end of file.
int <a href="#">read</a> (byte[ ] b)	It reads some number of bytes from the input stream and stores them into the buffer array b.
int <a href="#">read</a> (byte[ ] b, int m, int n)	It reads up to n bytes of data from the

	mth position of the input stream into an array of bytes.
void <a href="#">reset()</a> .	It moves the pointer to the beginning of the input stream
long <a href="#">skip</a> (long n)	It skips n bytes of data from the input stream.
void close()	It closes the current input stream

Eg: To read a file using Bytestream class

```
import java.io.*;
class Example13
{
    public static void main(String args[])
    {
        int c;
        try
        {
            FileInputStream f1= new FileInputStream("s1.dat");
            while (( c= f1.read())!=-1)
                System.out.print((char)c);
            f1.close();
        }
        catch(IOException e)
        {
            System.out.println("I/O error");
        }
    }
}
```

### OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. The methods in this class are given below:

Method	Use
void write(int)	It is used to write a byte to the current output stream
void write(byte[ ] b)	It is used to write an array of byte to the current output stream
void write(byte[ ] b ,int m, int n)	It writes n bytes from buffer array of data from the mth position.
void flush()	It flushes the current output stream.
void close()	It closes the current output stream

Eg: To create a file using Bytestream class

```
import java.io.*;
class Example11
{
    public static void main(String args[])
    {
        int c;
```

```

try
{
    FileOutputStream f1= new FileOutputStream("s1.dat");
    System.out.println("Enter text at end press ctrl+z");
    while (( c= System.in.read())!=-1)
        f1.write(c);
    f1.close();
}
catch(IOException e)
{
    System.out.println("I/O error");
}
}
}

```

### 5.3.5 Character streams

Java character streams are used to perform input and output operation on characters. Though there are many classes related to character streams but the most frequently used classes are **Reader** and **Writer**.

#### Reader class

Reader class is an abstract class .It is the superclass of all classes representing an input stream of characters. The methods in this class are given below:

Method	Use
int available()	It returns the number of characters that can be read from the current input stream.
int read()	It reads the next character of data from the input stream. It returns -1 at the end of file.
int read(char [] b)	It reads some number of characters from the input stream and stores them into the buffer array b.
int read(char[] b, int m, int n)	It reads up to n characters of data from the mth position of the input stream into an array of bytes.
void reset().	It moves the pointer to the beginning of the input stream
long skip(long n)	It skips n characters of data from the input stream.
void close()	It closes the current input stream.

Eg: To read a file using characterstream class

```

import java.io.*;
class Example14
{
    public static void main(String args[])
    {
        int c;
        try
        {

```

```

        FileReader f1= new FileReader("s2.dat");
        while (( c= f1.read())!=-1)
            System.out.print((char)c);
        f1.close();
    }
    catch(IOException e)
    {
        System.out.println("I/O error");
    }
}
}

```

### Writer class

Writer class is an abstract class. It is the superclass of all classes representing an output stream of characters. The methods in this class are given below:

Method	Use
void write(char)	It is used to write a character to the current output stream
void write(char[])	It is used to write an array of character to the current output stream
void write(char[] ,int m, int n)	It writes n bytes from character array of data from the mth position
void flush()	It flushes the current output stream.
void close()	It closes the current output stream

Eg: To create a file using characterstream class

```

import java.io.*;
class Example12
{
    public static void main(String args[])
    {
        int c;
        try
        {
            FileWriter f1= new FileWriter("s2.dat");
            System.out.println("Enter text at end press ctrl+z");
            while (( c= System.in.read())!=-1)
                f1.write(c);
            f1.close();
        }
        catch(IOException e)
        {
            System.out.println("I/O error");
        }
    }
}

```

## Summary

- ❖ Types of errors – Syntax ,Logical & Runtime errors
- ❖ Exception – Error at the run time of a program
- ❖ Handling exception -- using try catch block
- ❖ Multithreading -- Execution of more threads simultaneously
- ❖ Life cycle of thread – 5 states
- ❖ Different thread methods -- run(),start(),stop(),yield(),wait() etc
- ❖ Creating threads – using Thread class, Runnable interface
- ❖ Thread priority -- an integer number assigned by OS and used by the thread scheduler to decide when the thread should be allowed to run.
- ❖ Thread Scheduling -- process of allocating the CPU for execution of the threads
- ❖ Thread Synchronization -- capability to control the access of multiple threads to any shared resource
- ❖ Stream – path along which the data flows
- ❖ Types of streams – InputStream, OutputStream
- ❖ Stream classes – ByteArrayInputStream ,CharacterStream

**QUESTIONS**  
**PART - A (2 Marks)**

1. Define exception.
2. List the types of error.
3. What are the limitations of error handling?.
4. What is exception handling?.
5. What is the use of try block?.
6. Give the syntax of catch block.
7. What is the use of finally statement?.
8. Define multithreading.
9. What are the different ways to create a thread?.
10. List the states of thread.
11. Write any two thread methods and state their use.
12. Which method is used to get the priority of thread? Give its syntax.
13. Define thread scheduling.
14. What is thread synchronization?.
15. Define file.
16. What is stream?.List is types.
17. What is inputstream?.
18. What is outputstream?.

**PART - B (3 Marks)**

1. Explain the use of setPriority() method.
2. List the advantages of exception handling.
3. Explain about try block.
4. Explain about catch block.
5. Explain about finally block.
6. List the advantages of streams.
7. Write any methods of OutputStream class and state their use.
8. Write any methods of Reader class and state their use.

**PART - C (5/10 Marks)**

1. Explain in detail exception handling.
  2. Describe about thread methods.
  3. Explain the life cycle of thread with neat diagram.
  4. How you define and run a thread using Thread class?. Give example.
  5. How you define and run a thread using Runnable interface?. Give example.
  6. Explain about thread priority and thread scheduling.
  7. Write short notes on thread synchronization.
  8. Explain about Bytestream classes.
  9. Explain about Characterstream classes.
- .....