UNIT - I

Introduction about Java & OOP's

Object oriented Programming:

→ Object oriented Programming (OOP's) is a style of Programming, Characterized by the identification of classes of objects closely linked with the methods (functions) with which they are associated.

→ Object oriented Programming tries to map Code instructions with real world making the Code short and easier to understand.

→ OOPs Features:

→ Class :- Blue print of object
→ object :- Every object is instance of a class
→ Inheritance
→ Polymorphism
→ Encapsulation
→ Abstraction.

Java:-

Java is a popular programming language created in 1995 by James Gosling at sun Microsystem.

It is a simple programming language.

Java is an object-oriented programming language used in distributed environments on the internet.

it is a high-level language. That is also easy to read & understand. developers can "write once, run anywhere"; That means the compiled Java code will run on all Java compatible platforms without the need for recompilation. Java is a platform independent language. Java works on multiple platforms because when a Java program gets compiled, the compiler creates a .class byte code file that can run on any operating system that has the Java virtual Machine (JVM) installed on it. It is typically easy to install JVM on most major operating systems including ios.

Implementation of a Java application program involves

1. Creating program
2. Compiling program
3. Running the program

example :- simple.java

```
class Simple {
  public static void main(String args[])
  {
   System.out.println("Hello Java");
  }
}
```

O/P :- simple.java

"Hello Java"

OOPS Concepts :-

Object means a real world entity such as a pen chair, table, computer, watch etc. Object oriented Programming is a methodology or Paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts.

Object :-

Any entity That has state & behaviour is known as an object. For example Chair, Pen, table, keyboard, bike etc. It can be Physical or logical.

An object can be defined as an instance of a class

An object contains an address and takes up some space in memory.

objects can communicate without knowing the details of each others data (or) code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

eg:- A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating etc.

Class :-

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object.

Class doesn't consume any space.

Inheritance :-

when one object acquires all the properties and behaviors of a parent object, it is known as inheritence. It provides code reusability. It is used to achieve runtime polymorphism

for example. if sedan extends Car and Car extends for Vehicle, then sedan is also inherited from the Vehicle class.

Polymorphism :-

If one task is Performed in different ways. it is known as Polymorphism. For example: to convince the customer differently to draw something for example, shape, triangle, rectangle etc.

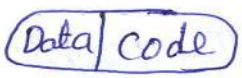
In Java we use method overloading and method overriding to achieve polymorphism

Another example can be to speak something.

a cat speaks meow, dog barks woof etc.

Encapsulation :-

Binding Code and data together into a single unit are known as encapsulation. For example it is wrapped with different medicines.



Abstraction :-

Hiding internal details and showing functionality is known as abstraction. For example phone call, we dont know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

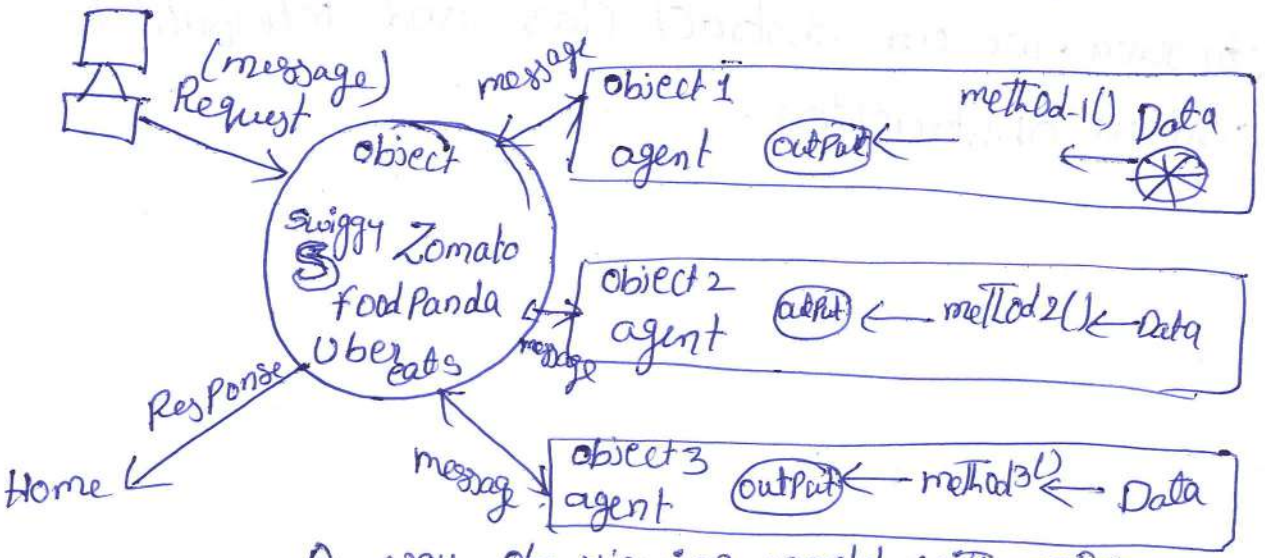
Object oriented Thinking:-

A way of of viewing woold :-

A way of viewing The woold is an idea to illustrate The object oriented programming Concept with an example of real woold situation

For example if we are in office and wish to get food to our family members who are at home from a hotel. Because of The distance from my office to home, There is no possibility of getting food from a hotel my self directly.

To solve This proble let me Call Zomato (an agent in food delivery Community), tell Them The variety and quality of food and The hotel name from which I wish to delever The food to my family members.



A way of viewing woold with OOPs

Agents and Communities :-

To solve my food delivery Problem, I used a solution by finding an appropriate agent (zomato) and pass a message containing my request. It is the responsibility of the agent (zomato) to satisfy my request. Here the agent uses some methods to do this. I do not need to know the method that the agent has used to solve my request This is usually hidden.

So in object-oriented programming, Problem solving is the solution to our proble which requires the help of many individual in the Community

An object oriented program is structures as a Community of interacting agents. Called objects. where each object provides a service (data & methods) that is used by other members of community.

In our example, the online food delivery system is a community in which the agents are zomato and set of hotels.

Each Hotel Provides a variety of Services that Can be used by other members like zomato, my self and my family in The Community.

Messages & Methods :-

To solve my problem, I started with a request to The agent Zomato, which led to still more requests among The members of The Community until my request has done. Here, The members of a Community interact with one another by making requests until The problem has satisfied.

In object-oriented Programming, every action is initiated by Passing a message to an agent (object) which is responsible for The action. The receiver is The object to whom The message, The receiver Performs some methods to Carry out The request Every message may include any additional information as arguments.

In our example, I Send a request to Zomato with a message That Contain food items, The quality of food, and The hotel details. The receiver uses a method to food get delivered to my home.

Responsibilities :-

In object-oriented Programming, behaviours of an object described in terms of responsibilities.
In our example my request for action indicates only The desired outcome (Food delivered to my family) The agent (Zomato) free to use any technique That solve my problem. By discussing a problem in terms of responsibilities increases The level of abstraction This enable more independence between The objects in solving complex Problems.

Classes and Instances :-

In object oriented Programming, all objects are instances of a class. The method invoked by an object in response to a message is decided by The class. all The objects of The class use The same method in response to a similar message.
In our example, The Zomato a class and all The hotels are sub classes of it. For every request (message) The class creates an instance of it and uses a suitable method to solve The problem.

Classes Hierarchies :- Inheritance

A graphical representation is often used to illustrate the relationships among the classes (objects) of a community. This Graphical representation shows that classes listed in a hierarchical tree-like structure. In this more abstract class listed near the top of the tree, and more specific classes in the middle of the tree, and the individuals listed near the bottom.

In object oriented Programming classes can be organized into a hierarchical inheritence structure. A child class inherits properties from the Parent class that higher in the tree.

Method Binding, overriding and Exception :-

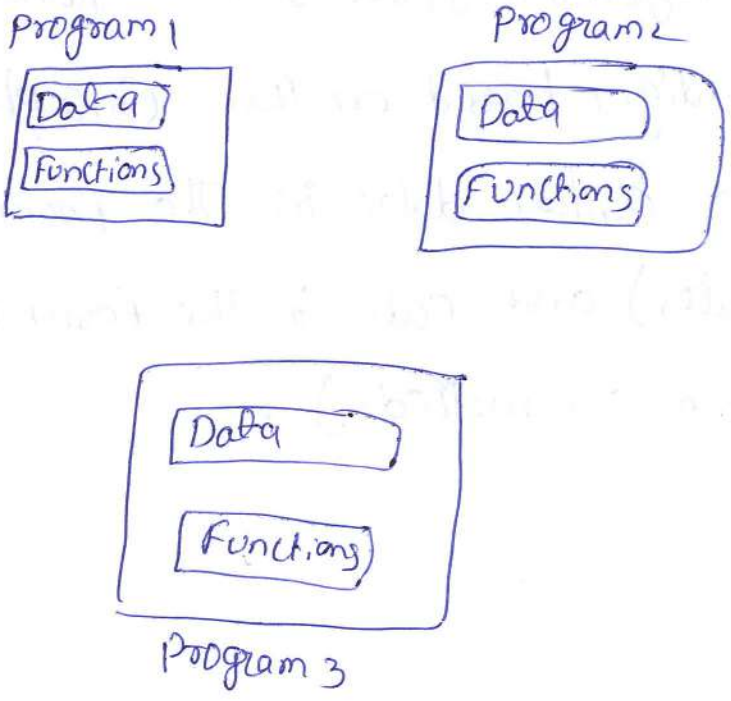
In the Class hierarchy both parent and child classes may have the same method which implemented individually. Here the implementation of the Parent is overridden by the child. or a Class may provide multiple definations to a single method to work with different arguments (overloading).

The search for the method to invoke in response to a request (message) begins with the class of this receiver. If no suitable method is found, the search is performed in the parent class of it. The search continues up the parent class chain until either a suitable method is found or the parent class chain is exhausted. If a suitable method is found, the method is executed. Otherwise, an error message is issued.

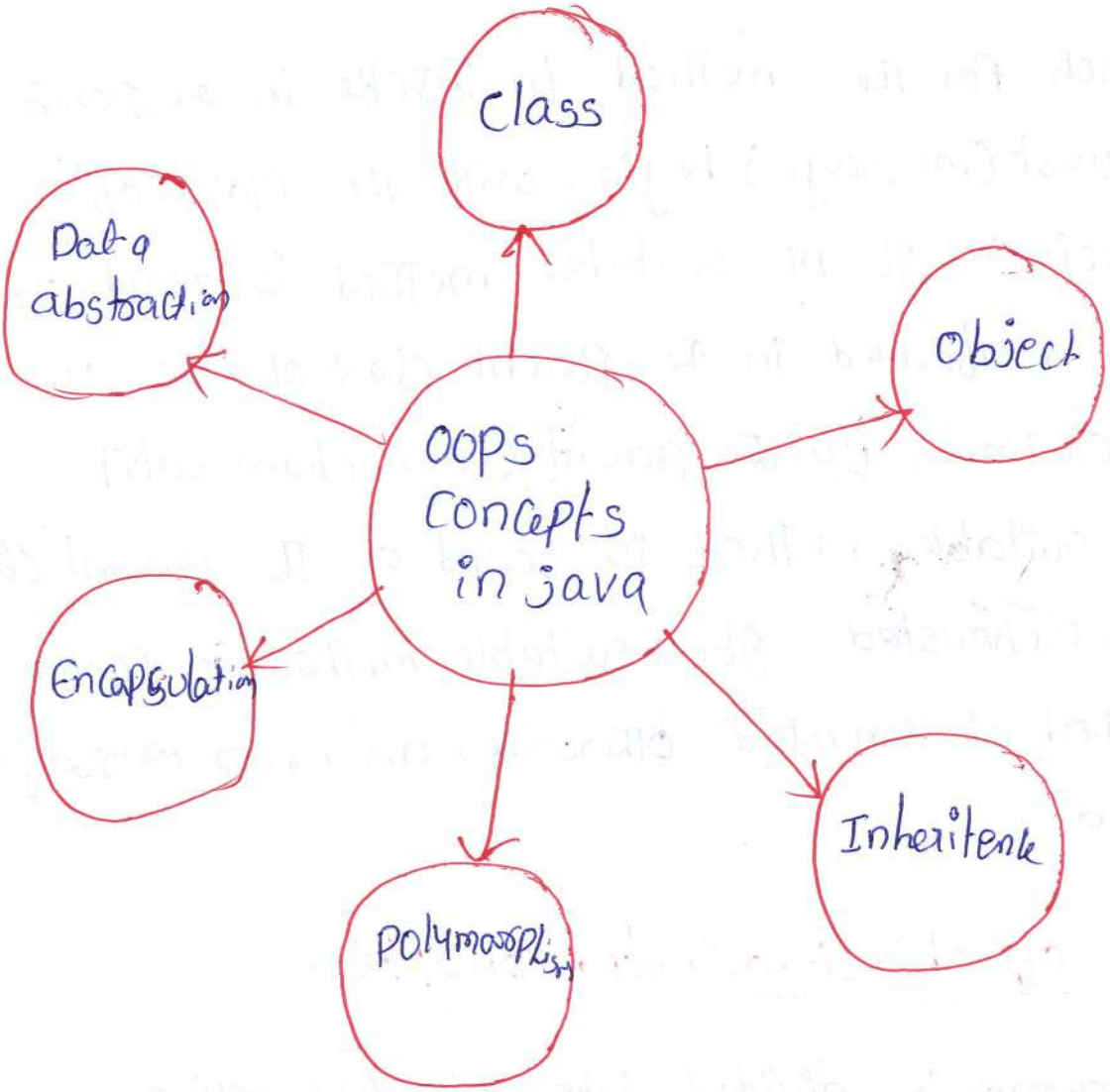
Summary of object-oriented concepts:-

→ Every program is divided into data & functions

→ Memory area is created for data & functions



→ Data & functions can be worked with the help of objects.



Object oriented Programming (OOPs) is a fundamental Programming Paradigm based on the Concepts of "objects" These objects can Contain data in the form of fields. (known as attributes) and code in the form of procedures (known as methods).

History of object orientence Programming :-

object oriented Programming is a paradigm That Focusses on The data and behaviour of The entities or objects That a Program manipulates. It organizes The code into classes, which are templates or blueprint for Creating objects. Classes define The attributes & methods of The objects, which are The data & functions That belongs to Them. objects Can interact with each other Through messages, which are Calls to Their methods. OOPS is based on The idea of bottom up design, where The problem is modeled by identifying The objects & Their Relationships. OOPS is often associated with languages like Java, C++ & Python.

origins of OOP :-

object oriented programming has its roots in The research and experiments of computer scientists in 1960s and 1970s. one of The pioneers of OOP was Ole-Johan Dahl, who along with Kristen Nygaard,

Developed the first object-oriented language "SIMULA" in 1967. SIMULA introduced the concepts of classes, objects, inheritance & polymorphism, which are the core features of OOP. Another person Alan Kay, who along with his colleagues at Xerox PARC, developed the language smalltalk in 1972. smalltalk was the first pure object-oriented language where everything was an object, even numbers & functions. small talk also introduced the concepts of dynamic typing, garbage collection, and Graphical User Interface.

Overview of Java :-

Java programming language was originally developed by Sun microsystems which was initiated by James Goslings and released in 1995 as core component of Sun Microsystem's Java Platform.

Java is,

Object-oriented :- In Java, everything is an object. Java can be easily extended since it is based on the object model.

Platform Independent :- Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine(JVM) on whichever platform it is being run on.

Simple :- Java is designed to be easy to learn. If you understand the basic concept of OOPs Java, it would be easy to master.

Secure :- With Java's secure feature it enables to develop virus-free, tamper free systems. Authentication techniques are based on public key encryption.

Architecture - Neutral :- Java Compiler generates an architecture-neutral object file format, which makes the Compiled Code executable on many processors, with the Presence of Runtime system.

Robust :- Java makes an effort to eliminate error prone situations by emphasizing mainly on Compile time error Checking & Runtime Checking.

MultiThreaded :- with Java's multiThreaded feature it is possible to write Programs that can Perform many tasks simultaniously.

High Performance :- with the use of Just In-time Compilers Java enables high Performance.

Distributed :- Java is designed for the distributed environment of the Internet.

Dynamic :- Java is Considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java Programming can Carry extensive amount of Run time information that Can be used to verify & resolve accesses to objects on Run-time.

Object-oriented design:-

Object oriented design (OOD) is the process of planning a system of interacting objects for the purpose of solving a software problem. It is a method for software design. By defining classes & their functionalities for their children, each object can run the same implementation of the class with its own state.

The OOP concepts was basically designed to overcome the drawback of structural & procedure programming structures programming methodologies, which were not so close to real world applications. The demand was increased but still conventional methods were used. This new approch brought a revolution in the programming methodology field.

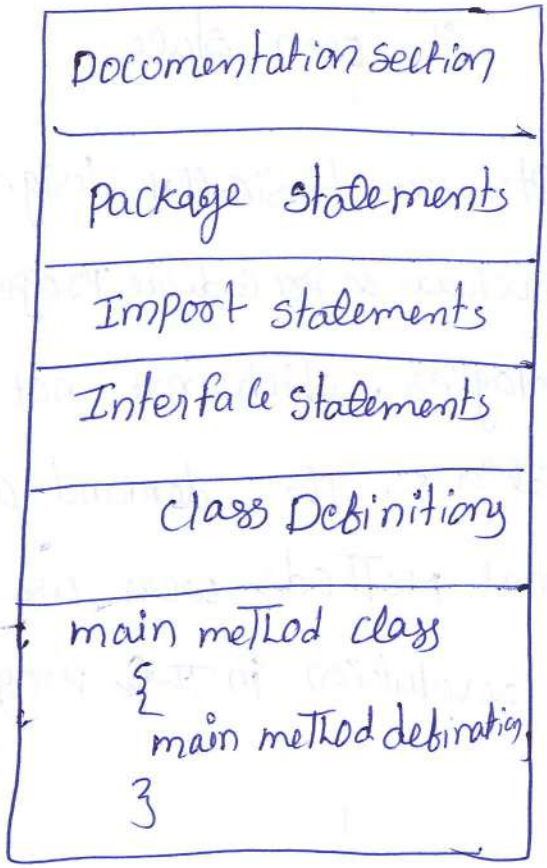
Object-oriented programming is nothing but that which allows. The writing of programs with the help of certain classes and real time objects.

This approach is very close to the real-world and its applications because the state & behaviour of these classes & objects are almost the same as realworld objects

Structure of Java Program:-

Java is an object oriented programming, platform independent & secure programming language that makes it popular.

In the Basic structure of Java program we have.



structure of Java Program

Documentation section :-

The Documentation Section is important section but it is optional in Java Program. It includes basic information about java program. It improves the readability of Program what ever we write in documentation section, the Java Compiler ignores the statements during the execution of the Program. To write the statements in documentation section, we use comments. The comments may be single line, multi-line and documentation comments

single line comments :-

It starts with a pair of foowording slash (//)

eg:- //first Java Program

Multi line comment :- It starts with a /* and end with */ we write between these two symbols.

eg:- /* It is an example of multiline comment */

Documentation comment :-

It starts with the delimiter (/**) and ends with */

eg:- /** It is an example of documentation comment */

Package Declaration:-

The Package Declaration is optional. It is placed just after the Documentation section. In this section we declare the Package name in which the class is the class is placed. Note that there can be only one package statement in java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different Packages & directories based on the module they are used. For all these classes Packages belongs to a single parent directory. We use the keyword "Package" to declare the package name

Package tkrcet ;//where tkrcet is the Package name

Import statements :-

The Package contains the many predefined classes & interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the "import" keyword to import the class.

It is written before the class declaration and after the package statement. we use the import statement in two ways, either import a specific class or import all classes of a particular package. In a java program we can use multiple import statements.

eg:- import java.util.Scanner ;// it imports the Scanner class only.
import java.util.* ;// it imports all the class of the java.util package.

Interface section:-

It is an optional section. we can create an interface in this section if required. we use the "interface" keyword to create an interface. An interface is a slightly different from the class. It contains only constants and methods declarations. Another difference is that it cannot be instantiated. we can use interface in classes by using the "implements" keyword. An interface can also be used with other interfaces by using the "extends" keyword.

eg:-

```
interface Car
{

  Void Start();
  Void Stop();

}
```

Class Defination:-

A Java Program may contain more Than one class defination. we use The "Class" keyword to define the Class. The class is a bluePrint of a Java Program. It Contains information about user-defined methods, variables and Constants. Every Java Program has at least one class that Contains The main() method.

eg:-

```
Class student //class definition
  {

  }
```

Class Variables and Constants :-

We define variables & constants that are to be used later in the Program. In a java program. The variables & constants are defined just after the class defination. The variables and constants store values of the parameters. It is used during the execution of the program. We can also decide and define the scope of variables by using the modifiers. It define the life of the variables

eg:-

```
Class Student // Class definition
{
    String Sname; //variable
    int id;
    double percentage.
}
```

Main Method class:-

In this section, we define the main() method. It is essential for all java Programs. Because the execution of all Java Programs starts from the main() method. In other words, it is an entry point of class. It must be inside the class. Inside the main method we create objects and call the methods.

We use following statements to define the main() method:

syntax:-

```
Public static void main (String args[])
 {

 }
```

eg:-

```
Public class Student
 {
Public Static Void main (String args[])
 {
 // statements
 }
 }
```

Methods & Behaviour :-

We define the functionality of the Program by using the Methods. The methods are the set of instructions that we want to perform. These instructions executes at runtime and perform the specified task.

```
Public class Demo
{
 Public static void main(String args[])
 {
 Void display()
  {
  System.out.println("welcome to tkrcet");
 }
 // Statements
 }
}
```

Java Buzz words :-

Java is the most popular object-oriented Programming language. Java has many advanced features, a list of key features is known as Java Buzzwords. The java Buzz words are;

→ simple
→ Secure
→ Portable
→ object oriented
→ Robust
→ Architectural-neutral (or) Platform Independent
→ multi Threaded
→ Interpreted
→ High Performance
→ Distributed
→ Dynamic

Java Datatypes :

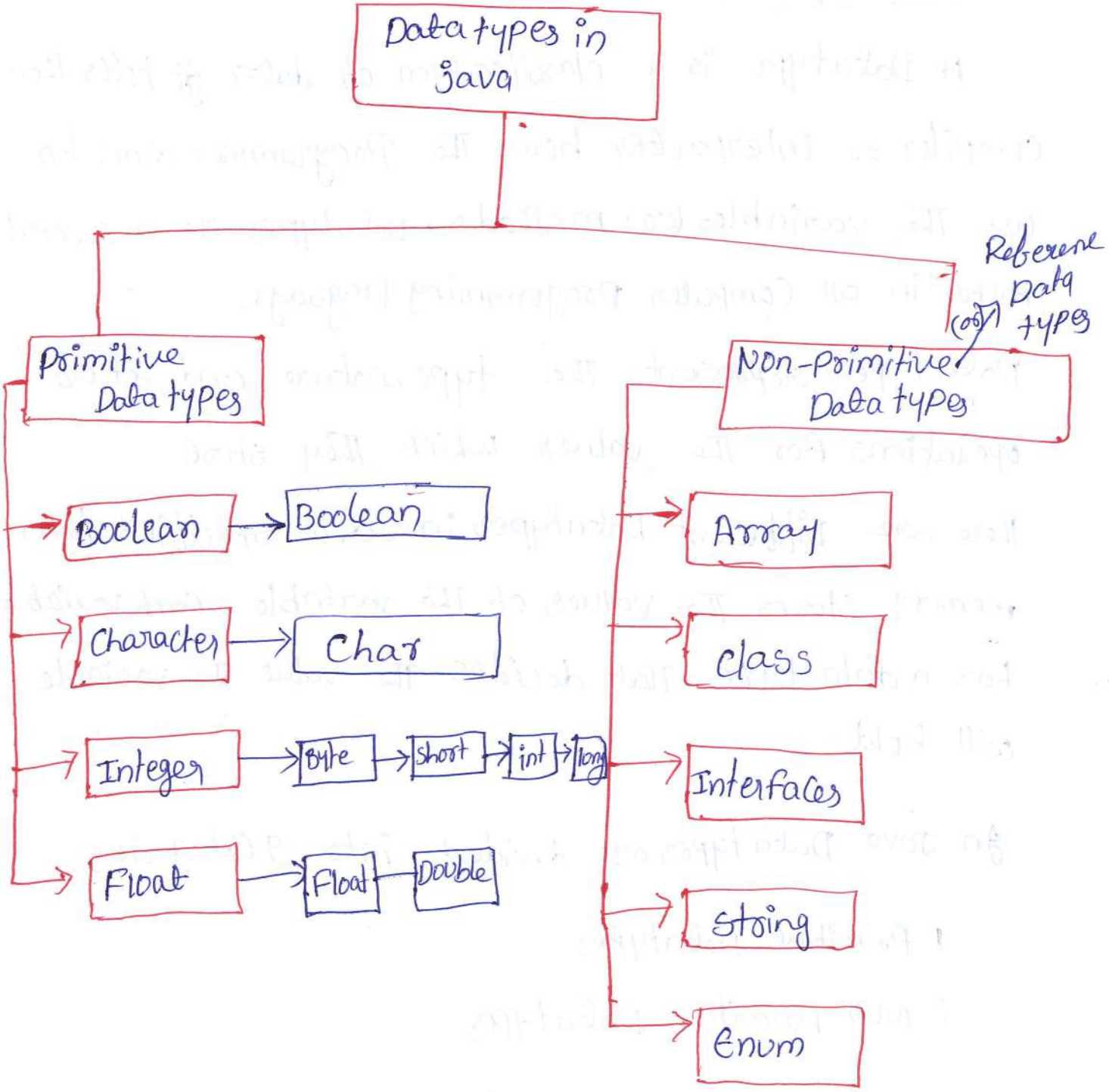
A Datatype is a classification of data it tells the compiler or interpreter how the programmer aims to use the variables (or) methods. Datatypes are a crucial factor in all Computer programming language.

Data type represents the type, nature, and set of operations for the values which they store.

There are Different Datatypes in Java specifies how memory stores the values of the variable. Each variable has a data type that decides the value the variable will hold.

In Java Datatypes are divided into 2 Categories.

1. Primitive Datatypes
2. Non-primitive Datatypes



Data types in Java

Primitive Data types :-

Primitive Datatypes specify the size & type of variable values. They are the building blocks of data manipulation and can not be further divided into simpler Datatypes

There are 8 Primitive Datatypes in Java - Boolean Char, byte, int, short, long, float & Double.

Boolean type :-

A Boolean Datatype can store either true or False. They can be used to check whether two values are equal or not. Programmers use it as a flag variable to track true or false conditions.

The default boolean value is false. The boolean type's size depends on the Java virtual Machine. Therefore it fluctuate on different Platforms

```
Class BooleanDataTypes
{
public static void main(String args[]) {
 boolean var1 = true;
 if (var1 == true) // Check if the value is true or false
 {
  System.out.println("Boolean value is true");
 }
 else
  {System.out.println("Boolean value is false");
 }
```

O/P :- Boolean value is true.

Character type :-

The Char datatype stores a single character. It stores lower case & upper case characters. which must be enclosed in single quotes. The Char Datatype in java supports unicode characters and provides Provision to multiple languages like English, French, German etc. It takes memory space of 16 bits (or) 2 bytes. The value stored range between 0 to 65536

Example :-

```
Class CharDataType {
  public static void main(String[] args) {
    Char var1 = 'A';
    Char var2 = 'd';
   System.out.println(var1);
   System.out.println(var2);
   }
}
```

O/P :-

```
A
d
```

Integer type:-

An integer type stores an integer number with no fractional (or) Decimal Places. Java has four Integer Types.

Byte:- The Byte is the smallest Data type among all the integer Datatypes. it is an 8-bit signed two's Complement integer. It stores whole numbers ranging from -128 to 127.

short:- short is a 16-bit signed two's Complement integer. it stores whole numbers with values ranging from -32768 to 32767. Its default value is 0.

int:- int is a 32-bit signed two's Complement integer that stores integer values ranging from -2147483648 (-2^{31}) to 2147483647 ($2^{31}-1$) Its default value is 0.

Long:- long is a 64-bit signed two's Complement integer that stores values ranging from -9223372036854775808 (-2^{63}) to 9223372036854775807 ($2^{63}-1$). It is used when we need a range of values more than those provided by int. Its default value is 0L. This data type ends with 'L' or 'l'.

eg:-

```
Class IntegerDataTypes
{
 Public Static Void main (String[]args){
   int a=10;
   Short s=2;
   byte b= 6;
   long L = 12536213322 3L;

System.out.println("The integer variable is "+a+'\n');
System.out.println("The Short variable is"+s+'\n');
System.out.println("The byte variable is"+b+'\n');
System.out.println("The long variable is"+l);
}
}
```

O/P:-

```
The integer variable is 10
The Short variable is 2
The byte variable is 6
The long variable is 12536213322 3
```

Float type:-

Floating-Point is used for expressions involving fractional precision. It has two types float & Double.

Float:-

It is a floating point data type that stores the values, including their Decimal precision. It is not used for Precise data such as currency or research data.

A Float value:-

→ is a single-Precision 32-bit or 4 bytes IEEE 754 floating point.

→ Can have a 7-digit decimal precision

→ ends with an f or F

→ default value = 0.0f

→ Stores fractional numbers Ranging from 3.4e-038 to 3.4e+038.

Double:-

The Double Datatype is similar to float. The difference b/w the two is that is double twice the float in the Case of decimal precision. It is used for decimal values just like float & should not be used for precise values.

A double value :-

→ is a double precision 64-bit (or) 8-bytes IEEE 754 floating-point

→ Can have a 15 digit decimal precision.

→ default value = 0.0d

→ stores fractional numbers ranging from 1.7e-308 to 1.7e+308.

Non-Primitive Datatypes :

Non-Primitive datatypes (or) Reference Datatypes refer to instances (or) objects. They cannot store the value of a variable directly in memory. They store a memory address of the variable. Unlike primitive datatypes are user-defined. Programmers create them and can be assigned with null. All non-primitive datatypes are of equal size.

Array:-

An Array Holds elements of the same type. It is an object in java, and the array name (used for dellaration) is a reference value that carries the base address of the continous location of elements of an array

eg:-

int Array_Name = new int[7];

String:- The string datatypes stores a sequence (or) array of characters. A string is a non-primitive data type, but it is predefined in java. String literals are enclosed in double quotes.

Class:- A class is a user defined data type from which objects are created. It describes the set of properties or methods common to all objects of the same type. It contains fields & methods that represent the behaviour of an object. A class gets invoked by the creation of the respective object.

There are two types of classes:- a blue print & template.

Interface:-

An interface is declared like a class. The key difference is that the interface contains abstract methods by default; they have nobody.

Enum:- An Enum similar to a class, has attributes and methods. However unlike classes, enum constants are public, static and final.

Developers can not use an enum to create objects and it can not extend other classes. But the Enum can implement interfaces.

Variables & Arrays :-

In Java variables are the data containers that save the data values during Java Program execution. Every variable in Java is assigned a datatype that designates the type and quantity of value it can hold. A variable is a memory location name for the data

Types of variables in Java :-

1. Local variables
2. Instance variables
3. Static variables

1. Local Variables :-

A variable defined within a block or method or constructor is called a local variable.

```
int a = 10;
```

2. Instance variable :-

Instance variables are non-static variables and are declared in a class outside of any method, constructor or block.

Static variable:-

Static variables are also called as Class variables These variables are declared similarly to instance variables. The difference is that static variables are declared using the static keyword with in a class out side of any method, constructor, or block. Static variable are created at the start of program execution and destroyed automatically when execution ends.

```
Class GFG
  {
    // static variable
     static int a;

   // instance variable
   int b;

}
```

Arrays:-

An array is a collection of similar type of elements which has contiguous memory location. Java array is an object which contains elements of a similar Datatype. The elements of an array are stored in a contiguous memory location. It is a Data structure where we store similar elements. we can store only a fixed set of elements in a java array.

Array in Java is index-based The first, The first element of The array is stored at The 0TH Index, 2nd element is stored on 1st index and so on.
Unlike C/C++, we can get The length of The array using The length member. In C/C++, we need to use The size of operator.

In Java array is an object of a dynamically Generated Class. java array inherits The object class and implements The serializable as well as cloneable interfaces. we can store Primitive values or objects in an array in java.

→ In Java, all arrays are dynamically allotted

→ Arrays may be stored in contiguous memory.

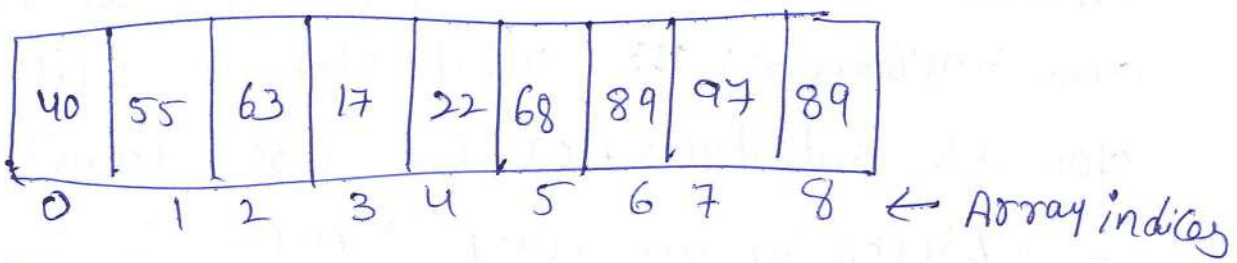
→ since arrays are objects in Java, we can find their length using. The object property length

→ A Java array variable can also be declared like other variables with [] after the data-type.

→ The variables in the array are ordered and each has an index beginning with 0

→ Java array can also be used as a static field a local variable, or a method parameter

An array can contain primitives (int, char, etc) and object references of a class depending on the definition of the array. In the case of primitive datatypes the actual values might be stored in contiguous memory locations.



Array length = 9
First Index = 0
last Index = 8

Define an Array :-

Arrays in Java are easy to define and declare, First we have to define the array.

```
type var-name[];
(or)
type[] var-name;
```

Here the type is int, String, Double (or) long. var-name is the variable name of the array

Declare An array :-

Types of Arrays :-

There are Three types of arrays. we use These types of arrays as per The requirment of The Program

1. One dimensional Array :-

Also known as a linear Array, The elements are stored in a single row

```
int Array = new int[5];
```

eg:-

```
Public Class Main {

    Public static void main(Stoing[] args) {
    int arr[] = {1,2,3,4,5};
    int sum = 0;
    for(int i=0; i<arr.length; i++) {

        sum += arr[i];
    }
    System.out.println("sum of the array is: "+sum);
}

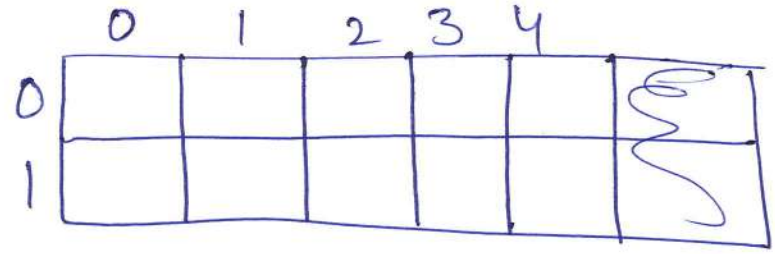
}
```

Two Dimensional Array :-

Two Dimensional arrays stores the data in rows & columns.

```
int[][] Array = new int[2][5];
```

In this array has two rows & five columns. The index starts from 0, 0 in the left upper cooner. to 1, 4 in the right lower cooner



In the java code, we have a, two dimensional array. we have two rows and three columns Brackets seperates the rows and the no. of elements seperates the columns. for this we use two loops. one for rows & one for each element in the row.

```
Public class Main {
  Public static void main (String [] args) {
   int arr[][] = { {1,2,3}, {4,5,6} };
   for (int i=0; i<2; i++) {
      for (int j=0; j<3; j++) {
        System.out.println( arr[i][j]+" ");
      }
   System.out.println();
   }
  }
}
```

MultiDimensional Array:-

This is the Combination of two or more arrays or nested arrays. we can even use more than two rows & columns.

Operators in Java:-

Java Provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Operators in Java are sybols used for performing specific operations in java.

Types of operators in Java:-

There are multiple types of operators in Java

1. Arithamatic operators:-

They are used to perform simple arithamatic operations on primitive Data types

addition, substraction, Multiplication, Divison, Modulo

2. Unary operators:

Unary operators need only one operand. They are used to increment, decrement (or) negate a value.

- : U. minus used for negating value
+ : U. plus indicate the positive value.

++ : Increment operator

Pre Increment

Post Increment

-- : Decrement operator.

Pre decrement

Post decrement

! : Logical Not operator. used for inverting a boolean value.

3. Assignment operator:-

'=' Assignment operator is used to assign a value to any variable. It has right-to left associativity. i.e value given on the right-hand side of the operator is assigned to the variable on the left & therefore right hand side value must be declared before using it or should be a constant.

The General format of assignment operator is

variable = value;

The assignment operator can be combined with other operators to build a shorter version of statement is called a compound statement.

+=, -=, *=, /=, %=

eg: a = a+5;

a+=5;

Relational operators:

These operators are used to check for relations like equality, greaterthan & lessThan.

Variable relational operator value.

==, !=, <, <=, >, >=

Logical operators:

These operators are used to perform logical functions

&&, ||, !

Logical AND, Logical OR, Logical Not

Ternary operators:

The ternary operators is a shorthand version of the if-else statement. It has three operands & hence the name ternary.

Condition ? if true : if false.

eg:- a=20 b=10 c=30, result

result = ((a>b) ? (a>c) ? a : c : (b>c) ? b : c);

Bitwise operators :-

These operators are used to Perfoom The manipulation of individual bits of a number. They Can be used with any of The integer types. They are used when performing update & query operations of The Binary indexed tree.

& - Bitwise AND operator :-

| - Bitwise OR operator :

^ Bitwise XOR operator :

~ Bitwise Complement operators :

Shift operators :-

These operators are used to shift The bits of a number left or Right Thereby multiplying or dividing The number by two respectively

<< Left shift operators

shift The bits of The numbers to The left and fills 0 on voids left as a result.

>> signed Right shift operators :

shift The bits of the numbers to the right and fill 0 on voids left as Result

>>> unsigned Right shift operator:

shifts the Bits of the numbers to the right & fills 0 on void left as a result the left most bit is set to zero.

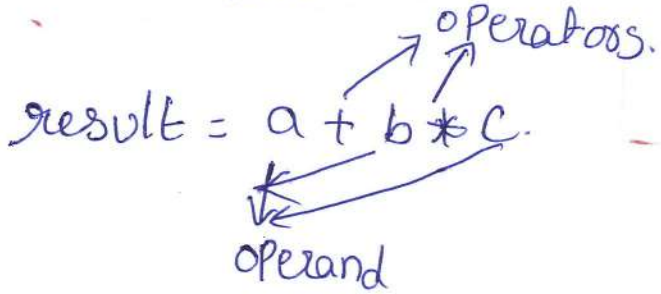
instance of operator:-

The instance of the operator used for type checking. It can be used to test if an object is an instance of a class, a subclass (or) an interface.

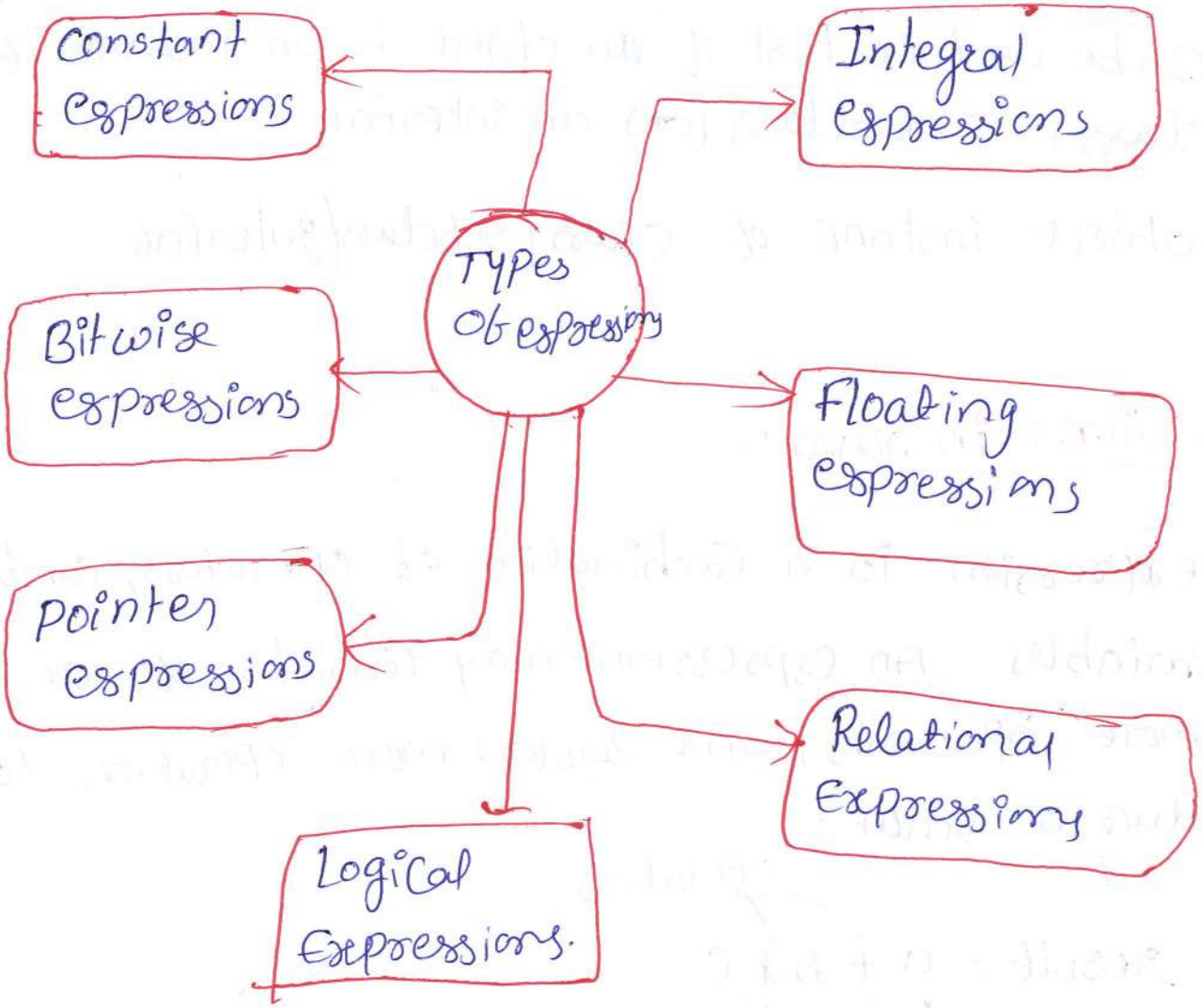
object instance of class / subclass / interface

Expressions in Java:-

An expression is a combination of operators, constants & variables. An expression may consists of one or more operands; and zero (or) more operators to produce a value.



Types of Expressions :-



Constant expressions:

Constant expressions Consist of only Constant values A Constant value one that doesn't change

eg:- 5, 10+5, 5/6.0, 'x'

Integral expressions:

Integral expressions are Those which produce integer results after implementing all the automatic & explicit type conversions

eg:- x, x*y, x+int(5.0)

where x and y are integer variables.

Floating expressions:

Float expressions are which Produce floating point results after implementing all automatic and explicit type conversions

x+y, 10.75

where x & y are floating Point variables.

Relational expressions:

Relational expressions yields results of type bool which takes a value true (or) False. when arithmatic expression are used on either side of a relational operator they will be evaluated first and then the result Compared

x<=y, x+y>z

Logical expressions:

Logical expression combine two or more relational expressions & produces bool type results

x>4 && x==10, x==10||4==5

Pointer expressions:-

Pointer expressions Produce address values

&x, ptr, ptr++

where x is a variable ptr is a pointer

Bitwise expressions:-

Bitwise expressions are used to manipulate data at bit level They are basically used for testing or shifting bits.

x<<3

shift three bit positions to left

y>>1

shift one bit position to right

Control statements:

Java compiler executes the code from top to Bottom. The statements in the code are executed according to the order appear.

Java provides statements that can be used to control the flow of Java code. Such statements called java controlled statements.

Java provides three types of control flow statements

1. Decision making statements
   - → if statement
   - → switch statement
2. Loop statements
   - → do while Loop
   - → while Loop
   - → for Loop
   - → for-each Loop
3. Jump statements
   - → break statement
   - → Continue statement

Decision Making Statement:

Decision making statements decide which statement to execute and when.

There are two types of Decision making statements

→ if statement

→ switch statement

1) if statement:

In java if statement is used to evaluate a Condition The control of the program is diverted depending upon the specific condition. In java there are four types of if-statements

1. simple if statement:

syntax:
```
if(condition)
  {
   statement 1;
  }
```

2. if else statement:

syntax:
```
if (condition)
  { statement 1;
  }
else
  { statement 2;
  }
```

3. if-else-if ladder :-

Syntax:
```
if (condition 1) {
    statement 1;
}
else if (condition 2) {
statement 2;
}
else {
statement 3;
}
```

4. Nested if Statement :

syntax:
```
if (condition 1)
    {
    statement 1;
  if (condition 2)
    {
    statement 2;
  }
else {
  statement 3;
  }
}
```

Switch statement:

Switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched.
The switch statement is easier to use insted of if-else-if statements.

Syntase:

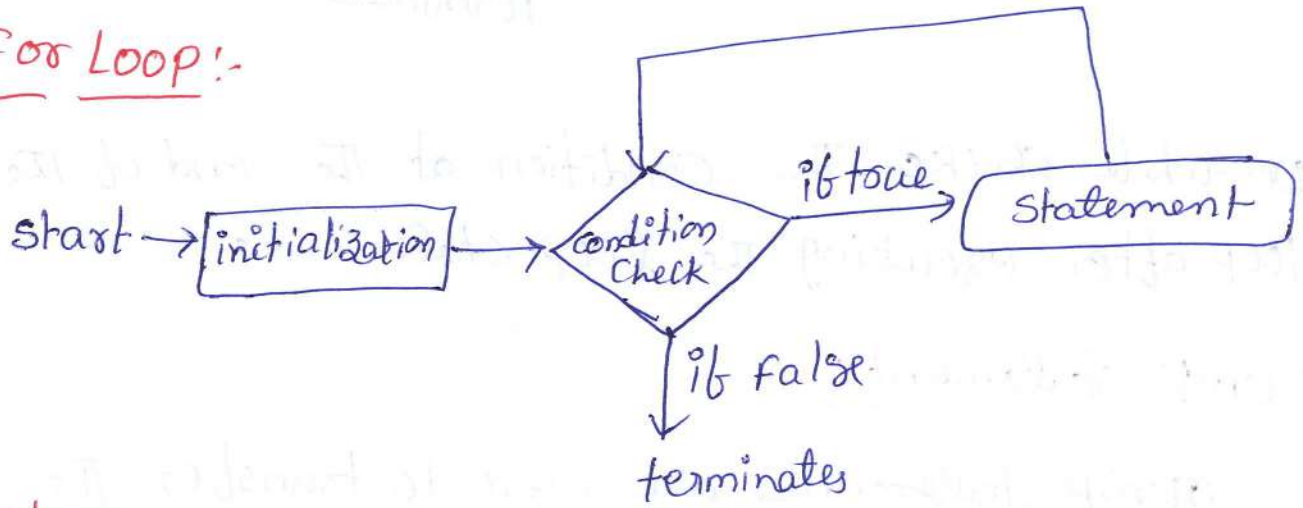
```
switch(espression){
 case:value 1;
  Statement 1;
  break;
  ;
  ;
  ;
  ;
 Case value N;
  Statement N;
  break;
 default;
 default Statement
}
```

Loop statements:

Loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

1. for loop
2. while Loop
3. do while Loop

For Loop:-

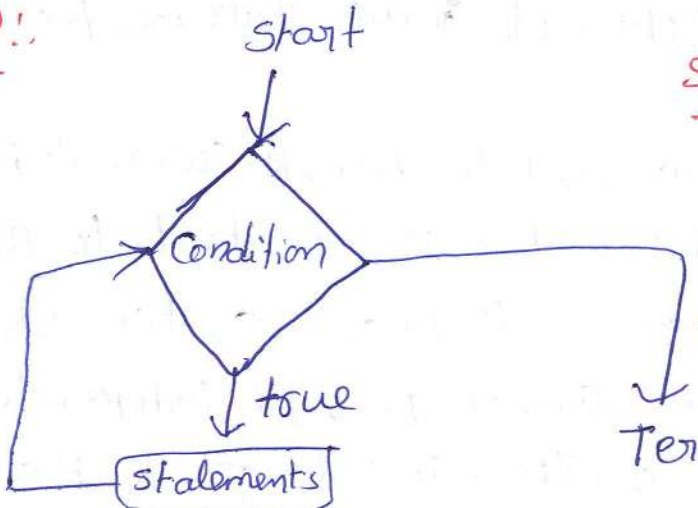


Syntax:-

```
for(initilization, condition, increment/decrement) {

}
```

while LOOP:-



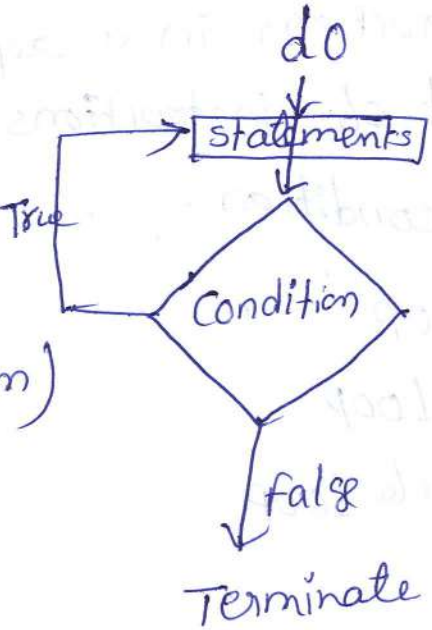
syntax:-

```
while (condition)
{
}
```

Do-while Loop:

syntax:

```
do
{
 // Statements
} while (Condition)
```



Do-while checks the condition at the end of the loop after executing the loop statements.

Jump Statements:

Jump statements are used to transfer the control of the program to the specific statements. In other words jump statements transfer the execution control to the other part of the program.

There are two types of jump statements

Break:

Break statement used to break current flow of the program and transfer the control to the next statement outside a loop(or) switch statement

break statemen cannot be used independently in java it can be written inside the loop(or) switch.

Continue statement:

Continue statement doesn't break the loop, whereas it skips the specific part of the loop & jumps to the next iteration of the loop immediatly

Introducing Classes :-

In Java, classes & objects are basic concepts of OOPS that are used to represent real world concepts and entities. The class represents a group of objects having similar properties and behaviour. For example the animal type dog is a class while a particular Dog named tommy is an object of the DOG class

Java classes :-

A class in Java is a set of objects which shares common characteristics / behaviours & common properties / Attributes. It is a user defined blue print or prototype from which objects are created. For example student is a class while a particular student named ravi is an object.

Properties of Java Class :-

1. Class is not a real world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory
3. Class is a group of variables of different datatypes and a group of methods
4. A class in Java can contain
   - → Data member
   - → Method
   - → constructor
   - → Nested class
   - → Interface.

eg:

```
Class Student {
    // data member (also instance variable
     int id;
    // data member
    String name;
Public static void main (String args[])
 {
```

~~public static void main(~~

```
// Creating an object of Student
Student S1 = new Student();
System.out.println(S1.id);
System.out.println(S1.name);
}
```

Components of Java classes :-

1. **Modifiers** :- A class can be Public or has default access
2. **Class keyword** : Class keyword is used to create a class
3. **Class name** : The name should begin with an initial letter (capitalized by convention)
4. **Superclass (if any)** :

   The name of the classes Parent
5. **Interfaces (if any)** :

   A Comma-Seperated list of interfaces implemented by the class, if any preceded by the keyword implement

   A Class can implement more than one interface
6. **Body** :- The class body surrounded by braces { }

Methods and Classes:

The Method in Java (or) Methods in Java is a collection of statements that perform some specific task and return the result to the caller. A Java method can perform some specific task without returning anything Java methods allow us to reuse the code without retyping code.

In Java every method must be part of some class. That is different from languages like C, C++ & Python

1. A method is like a function i.e used to expose the behaviour of an object
2. It is a set of codes that perform a particular task.

Advantage of Methods:

Code reusability
Code optimization

1

Types of methods :-

1. Predefined method :

Predefined methods are the methods that is already defined in the java Class libraries is known as predefined methods . It is also known as standard library. method (or) built in method .
we can use these methods just by calling them in the program at any point .

2. User defined method :-

The method written by the user or programmer is known as a user defined method . These methods are modified according to the requirement .

2 ways to create a method in Java.

There are two ways to create a method in Java.

1. Instance method :-

Access the instance data using the object name . Declared inside a class .

```
// Instance method
void method name () {
    body // instance area
}
```

2. Static method :-

Access the static data using class name. declared inside the class with static keyword.

eg:- static void method_name() {
body // static area
}

Method signature :-

It Consist of the method name & a parameter list the return typ and exceptions are not Considered. as part of it

max(int x, int y) Number of parameters is 2

Name a method :-

A method name is typically a single word that should be a verb in lowerCase or a multi-word that begins with a verb in lower Case followed by an adjective, noun. after the first word the Fist letter of each word should be Capitalized.

Rules to name a Method :-

→ while defining a method, remember that the method name must be a verb and start with a lowercase letter.

→ If the method name has more than two words the first name must be ~~verb~~ followed by an adjective (or) noun.

→ In the multiword method name, the first letter of each word must be in upper case except the first word.

eg: findSum, ComputeMax, setX and getX

Method Calling :-

The methods need to be called for use its functionality. There can be three situations when a method is called.

A method returns to the code that invoked it when

→ It completes all the statements in the methods

→ It reaches a return statement

→ Throughs an exception.

eg:-

```
import java.io.*;
// Class 1
Class Addition {

    int sum = 0;

    // Method
    // To add two numbers
  Public int addTwoInt (int a, int b)
   {
     // Adding two integer value
       sum = a+b;

    // Returning summation of two values
     return sum;
  }
}
// Class 2
Class GFG {
     // Main driven Method
  public static void main (String[] args)
    {
```

```
// Creating object of class 1 inside main() method
    Addition add = new Addition();
    // Calling method of above class
    // to add two integers using instance created
    int s = add.addTwoInt(1, 2);
    // Printing the sum of two numbers.
    System.out.println("sum of two integer values:"+s);
}
}
```

o/p:- sum of two integer values: 3

String Handling :-

A string is a collection of characters. In Java a string is an object that represents a collection of objects. A string is a predefined class used to create string objects. It is an immutable object, which means it cant be updated once created.

The string class has a set of built-in methods.

charAt(): It Returns a character at a specific position.

equal(): It compares the two given Strings and return a Boolean that is True (or) False.

Concat(): Appends one string to the end of another

length(): Returns the length of specific string.

toLowerCase(): Converts the string to Lower Case letters

toUpperCase(): Convert the string to upperCase letters.

indexOf(): Returns the First Found Position of a Character.

substring(): Extracts the substring based on index values, Passed as an argument.

Inheritance:-

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of object oriented programming system

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods & fields of the parent class. You can add new methods and fields in your current class also

Syntax:-

```
class Subclass_name extends Superclass name
{
  // methods & fields
}
```

The extend keyword indicates that you are making a new class that derives from an existing class. In the terminology of Java, a class which is inherited is called a parent (or) super class and the new class is called child or sub class.

eg:-

```
class Employee {
 float salary = 40000;
}
class Programmer extends Employee {
  int bonus = 10000
 public static void main(String[] args){
  Programmer p= new Programmer();
System.out.println("Programmer salary is:" +p.salary);
System.out.println("Bonus of Programmer is:" +p.bonus);
 }
}
```

o/p:-

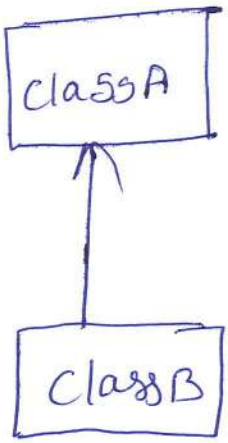
```
Programmer salary is : 40000.0
Bonus of Programmer is : 10000
```

Types of inheritance :-

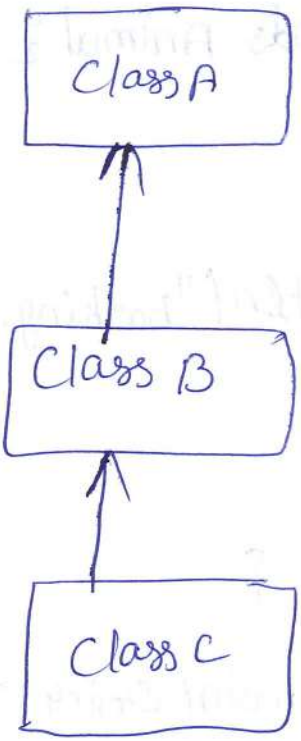
There can be Three types of inheritance in java.

→ single inheritance
→ multi level inheritance.
→ hierarchical inheritance

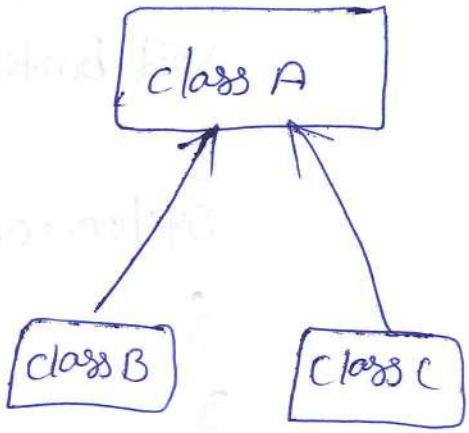
In java multiple & hybrid inheritances is supported through interface only.



single



Multilevel



Hierarchical.

Single Inheritance :-

when a class inherits another class, it is known as a single inheritance.

eg:-

```
Class Animal
{
 void eat()
 {
  System.out.println("eating");
 }
}
Class Dog extends Animal {
 Void bark()
 {
 System.out.println("barking...");
 }
}
Class Test Inheritance {
 Public static void main(String[] args) {
 Dog d = new Dog();
 d.bark();
 d.eat();
 }
 }
```

O/P:-

barking
eating.

Multilevel Inheritance:-

When There is a Chain of inheritance is known as multilevel inheritance.

eg:-

```
Class Animal {
    void eat()
    {
    System.out.Println("eating.... barking....");
    }
}
Class Dog extends Animal {
Void bark()
{
System.out.Println("barking...");
}
}
Class BabyDog extends Dog
{
void weep()
{System.out.Println("weeping...");
}
}
```

```
Class TestInheritance2 {
 Public Static void main (Sting args [])
 {
  BabyDog d = new BabyDog ();
    d.weep();
    d.bark();
    d.eat();
  }
}
```

o/p:-

```
weeping ....
barking ....
eating ....
```

Hierarchical Inheritance:

When two or more classes Inherits a single class it is known as hierarchical inheritance.

eg:-

```
Class Animal {
  void eat()
  {
  System.out.Println("eating...");
  }
}
class Dog extends Animal {
  void bark() {
  System.out.println("barking...");
}
}
class Cat extends Animal {
  void meow()
  {
  System.out.println("meowing...");
  }
}
class Test Inheritance3 {
  Public Static void main (String[] args).
  {
    Cat C = new Cat();
      C.meow();
      C.eat();
  }
}
```

o/p:-
meowing...
eating...

Member Access :

An access level determines whether other classes can use a particular member variable or call a particular method. The Java Programming language supports four access specifiers for member variables and methods.

private, protected, public, and, if left unspecified, package private.

Level	Who Can Access	Rule
private	no access to the outside world	If a class member is declared private it is accessible only inside the class that declared it. A private class member is not inherited by subclasses of that classes
Public	everyone	A subclass inherits all public members of its superclass
Protected	subclasses	A Protected class member is accessible inside the body of a subclass from the same package as the class or in a different package.

default (oo)
Package-level access

Package

Package-level class member is inherited only if the superclass & subclass are in the same package.

boo example.

Employee class has three members: a name field, a getName() method, and a setName() method.

The name field has been declared private and hence it is not accessible inside the Manager class because it is not inherited.

The getName() and setName() methods have been declared public and they are accessible from anywhere including manager class.

```
Class Employee {
    Private String name = "Unknown";
    public void setName (String name) {
        This .name = name;
    }
    Public string getName() {
        return name;
    }
}
```

```
class Manager extends Employee {
}
Public Class Main {
  Public static void main(Sting[] args) {
  // Create an object of the Manager class
    Manager mgr = new Manager();
  // set the name of the manager
    mgr.SetName("Mary");
  // Get the name of the manager
    Sting mgrName = mgr.getName();
  // Display the manager name.
   System.out.println("Manager name:" + mgrName);
    Employee emp = new Employee();
      emp.SetName("Tom");
    Sting empName = emp.getName();
    System.out.println("Employee name:" +empName);
  }
}
```

o/p:-

Manager name: Mary

Employee name; Tom

Constructors :-

Java constructors (or) constructors in Java is a terminology used to construct something in our Programs. A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor memory for the object is allocated in the memory.

It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

eg:-

```
import java.io.*;
class Geeks {
    // Constructor
    Geeks()
    {
        super();
        System.out.println("Constructor called");
    }
    // main function.
    Public static void main(String[] args)
    {
        Geeks geeks = new Geeks();
    }
```

Differences between java Constructors & java methods:

→ Constructors must have the same name as the class with in which it is defined it is not necessary for the method in java.

→ Constructors do not return any type while methods have the return type (or) void if does not return any value.

→ Constructors are called only once at the time of object creation while methods can be called any number of times.

Types of Constructors:-

There are three types of Constructors in Java

→ Default Constructor

→ Parameterrized Constructor

→ Copy constructor.

Default constructor :- A Constructor that has no parameter is known as default the Constructor. A default Constructor is invisible. And if we write a constructor with no arguments the compiler does not create a default Constructor. It is taken out. It is being overloaded and called a parameterized Constructor. The default Constructor changed into the Parameterized constructor. but parameterized Constructor can't change the default Constructor.

2. Parameterized Constructor in java:-

A Constructor that has Parameters is known as Parameterized Constructor. If we want to initialize fields of the class with our own values then use a Parameterized Constructor.

3. Copy Constructors in java:-

Unlike other Constructors Copy Constructor is passed with another object which copies the data available from the passed object to the newly created object.

eg: create a default Constructor.

```
Public class Dog {
    //default Constructor
    Public Dog(). {
        System.out.println("A new Dog is born");
    }
}
Public class Main{ }
    Public static void main(String[] args)
        Dog myDog = new Dog();
    }
}
```

O/P:- A new Dog is born.

eg:-

Parameterized Constructors

```
Public Class Dog {
    String name;
    // Parameterized Constructor
    public Dog(String name) {
        this.name = name;
    System.out.println("A dog named " + name + " is born");
    }
}
Public class Main {
    Public static void main(String[] args) {
        Dog myDog = new Dog("Rex");
    }
}
```

O/P:-

A dog named Rex is born.

eg:- Copy Constructors.

```
Public Class Dog {
    String name;
    //Copy Constructor
    Public Dog(Dog originalDog) {
        this.name = originalDog.name;
        System.out.println("A copy of "+name+" is born!");
    }
}
Public class Main {
    Public static void main(String[] args) {
        Dog originalDog = new Dog("Rex");
        Dog CopiedDog = new Dog(OriginalDog);
    }
}
```

O/P:-

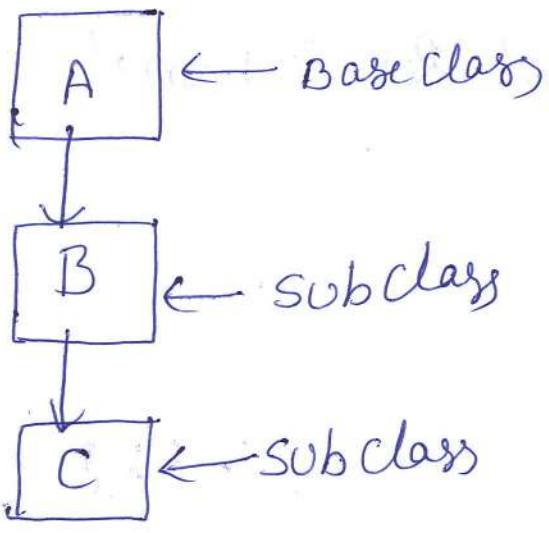
'A Copy of Rex is born!'

Creating Multilevel Hierarchy:

Inheritance is one of the most important Concepts of java programming, and it affects the way in which we design and write our java classes. It is through inheritance that a class can immediatly inherit the properties of another class. Java multilevel Hierarchy allows you to inherits properties of a grandparent in a child class.

In simple inheritance, a sub class (or) derived class derives its properties from its parent (or) superclass But in multilevel inheritance a sub class is derived from a derived class.

Therefore, in multilevel inheritance a ladder is formed which at each step increses by one and the lowermost class will have the properties of all the superclass.



Java multilevel hierarchy is highly effective in implementing Code reuse. It reduces programming efforts by using the predefined and pre-compiled classes. The readability of programs increases giving opportunity to a programmer to extend his code with additional features and methods on those offered by Parent and grand Parent classes.

eg:-

```
Class Stu.
 {
  int rollno;
  String name;
  Stu (int r, String n)
   {
     rollno = r;
     name = n
   }
   Void Show()
   {
  System.out.println("Student rollno:" +rollno);
  System.out.println("student Name:"+ name);
 }
}
```

```
Class Marks extends Stu
 {
   int  s1, s2, s3, s4, s5, sum1;
  marks(int r, String n, iht m1, int m2, int m3
    int m4, int m5)
    {
       super (r, n)
       s1 = m1;
       s2 = m2;
       s3 = m3;
       s4 = m4;
       s5 = m5;
    }
    void  showmarks()
     {
       show();
       Sum1 = s1+s2+s3+s4+s5;
  System.out.println("Total marks = "+sum1);
   }
}
class Percent extends marks.
 { float per;
  Percent (int r, String n, int m1, int m2, int m3,
        int m4, int m5 float P)
```

```
{
 super(r,n,m1,m2,m3,m4,m5);
   Per=P;
 }
 Void showper()
 {
  Showmarks();
  Per = Sum1/5;
 System.out.println("Percentage =" + Per);
 }
}
Class multieg
 { Public static void main (String[] args)
  {
   Percent P = new Percent(101, "Nancy", 90,75,85,90,80,0);
       P.showper();
  }
 }
```

O/P:

```
Student Roll No - 101
Student Name - Nancy
Total marks - 420
Percentage - 84.0
```

Super Keyword in Java:

The super keyword in Java is a reference variable which is used to refer immediate Parent class object. Whenever you create the instance of subclass, an instance of Parent class is created implicitly which is referred by super reference variable.

Usage of Java super keyword:-

1. super can be used to refer immediate Parent class instance variable.
2. super can be used to invoke immediate Parent class methods.
3. super() can be used to invoke immediate Parent class constructor.

1. super is used to refer immediate Parent class instance variable

We can use super keyword to access the data member or field of Parent classes. It is used if Parent class and child class have same field.

```
Class Animal {
 String Color = "white";
}
Class Dog extends Animal {
  String Color = "black";
 Void PrintColor() {
 System.out.println(Color);
 System.out.println(super.color);
}
}
Class TestSuper1 {
 Public static void main(String args[]){
  Dog d = new Dog();
  d.PrintColor();
 }
}
```

o/p:-

```
black
white
```

2. Super can be used to invoke Parent Class method.

The super keyword can also be used to invoke Parent Class method. It should be used if subclass contain the same method as Parent Class. In otherwords, it is used if method is overridden.

```
Class Animal {
void eat()
{
System.out.println("eating....");}
}
class Dog extends Animal
{
void eat()
{
System.out.println("eating bread...");
}
void bark()
{
System.out.println("barking...");
}
void work()
{
super.eat();
bark();
}
}
```

```
Class TestSuper2{
  Public static void main(String args[])
  {
   Dog d = new Dog();
    d.work();
  }
}
```

O/P:- eating ....
barking ....

3. Super is used to invoke Parent Class constructor

The super keyword can also be used to invoke the Parent class constructor.

eg:

```
Class Animal {
    Animal()
    {
   System.out.println("animal is created");
    }
}
Class Dog extends Animal
  {
   Dog()
   {
    super();
   System.out.Println("dog is Created");
   }
}
```

```
Class Testsuper3 {
  Public static void main(String args[])
  {
   Dog d = new Dog();
    }
}
```

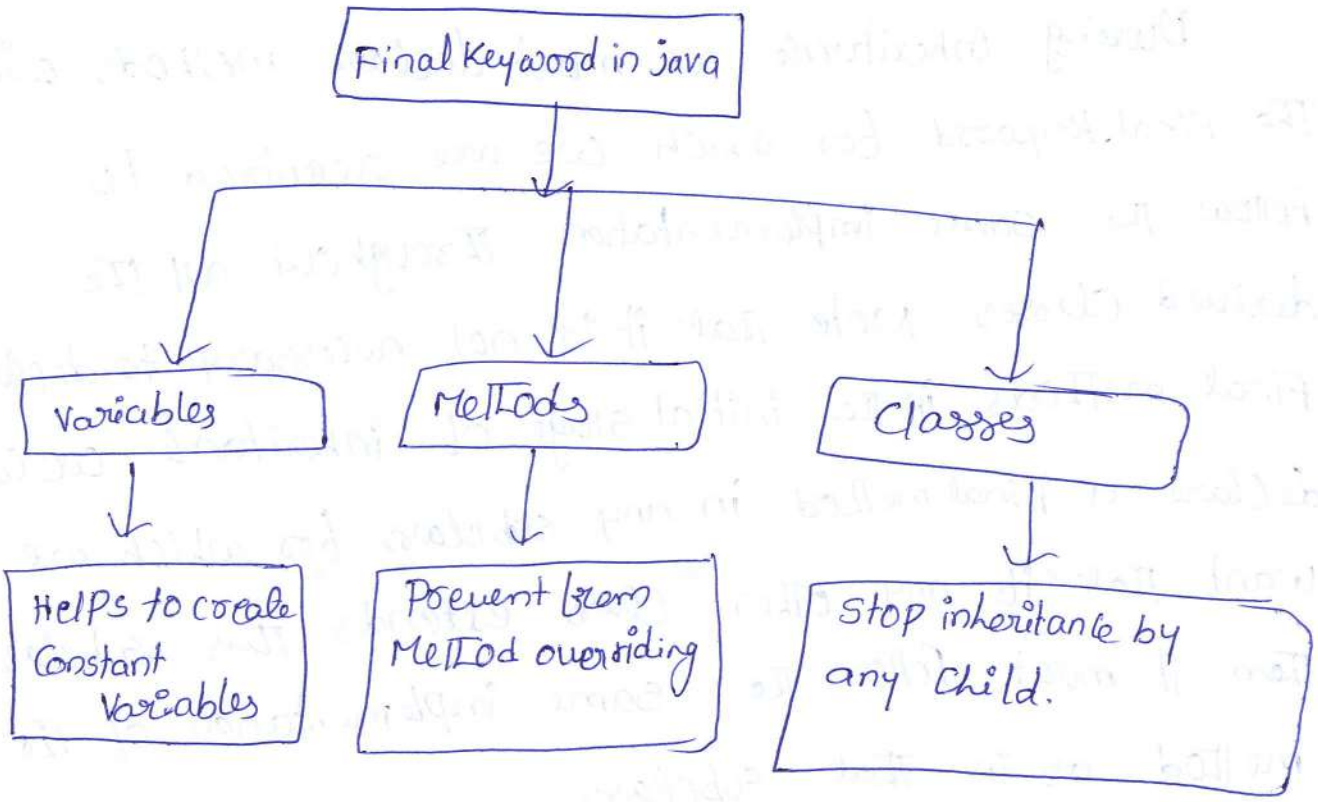
O/P:-

animal is created
dog is created

Using final with Inheritance in Java:

During Inheritance, we must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes. Note that it is not necessary to declare final methods in the initial stage of inheritance. We can declare a final method in any subclass for which we want that if any other class extends this subclass then it must follow the same implementation of the method as in that subclass.

→ The Final keyword is final that is we cannot change
→ We can use final keywords for variables, methods and class.
→ If we use the final keyword for the inheritance that is if we declare any method with the final keyword in the base class so the implementation of the final method will be the same as in derived class.
→ We can declare the final method in any subclass for which we want that if any other class extends this subclass



Variable with Final Keywords :-

When a final keyword is used with a variable it makes the variable Constant. Hence once assigned the value of the variable Cannot be changed.

```
Class Circle
{
Public static void main(String[] args) {
  Static final double PI = 3.14; // Constant variable to store
                                    The value of PI
    // Radius of the circle
      int radius = 4;
  // display the area of the circle
System.out.println("Area of Circle:" + (PI * radius * radius));
}
}
```

O/P :-
Area of Circle : 50.24.

Error Program:-

```
Public class Myclass {
Public Static void main (String[] args) {
Final int num = 10;
System.out.println (num);
  num = 20;
 }
}
```

o/p:- Error

Methods with Final Keyword :-

The Final keyword can also be used to declare a method as final. A final method cannot overridden by a subclass.

eg:-

```
Class Animal {
    final void displayCharacteristics() {
Static final int NUM_LEGS = 4;
Static final int NUM_EARS = 2;
Static final int NUM_Eyes = 2;
Static final int HAS_TAIL = 1;
```

```
System.out.println("General Characteristics of an animal are");
    System.out.println("Legs" + NUM_LEGS);
    System.out.println("Eyes" + NUM_Eyes);
    System.out.println("Ears" + NUM_EARS);
    System.out.println("Tail" + (HAS_TAIL==1?"yes":"NO"));
  }
}
Class wolf extends Animal {
  final void displayAdditionalCharacteristics() {
    System.out.println();
    System.out.println("Additional Characteristics:");
    System.out.println("sound: howl");
  }
  Public static void main (String[] args) {
    Wolf w = new wolf();
    w.displayCharacteristics();
    w.displayAdditionalCharacteristics();
  }
}
```

O/P:- General characteristics of an animal are

Legs : 4
Eyes : 2
Ears : 2
Tail : yes

Additional Characteristics
Sound: Howl

Final Keyword with Class:

Classes followed by the final keyword in java cannot be inherited by any subclass.

```
final Class MyClass {

}

Public class Mysubclass extends MyClass {
}
```

error will be occured.

eg:-

```
final class Day {
    static final int MINUTES-PER-DAY = 1440;
    static final int HOURS-PER-DAY = 24;
    static void displayHoursAndMinutes() {
    System.out.println("No.of Hours in a day" + HOURS-PER-DAY);
    System.out.println("No.of Minutes in a day" + MINUETS-PER-DAY);
    }
}
class Main{
    Public static void main(String[] args) {
        Day.displayHoursAndMinutes();
    }
}
```

O/P:
Number of hours in a day: 24
Number of minutes in a day: 1440

Polymoophism:

Polymorphism is derived from two Greek words, "poly" and "morph" which mean "many" & "forms respectively. Polymoophism meaning in java refers to the ability of objects to take on many forms

There are two types of polymoophism in Java.

→ Compile-time polymoophism

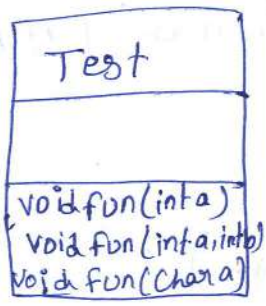
→ Runtime polymorphism

we can Perform Polymorphism in java by method overloading and method overriding.

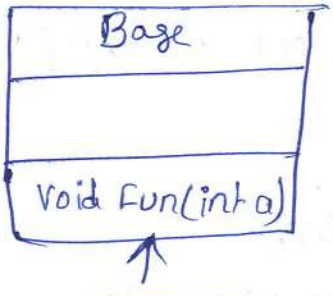
If you overload a static method in java it is the example of Compile time Polymorphism

Compile time Polymorphism :-(or) Adhoc Polymoophism :-

It is also known as static Polymorphism. This type of polymorphism is achieved by function overloading (or) operator overloading.



overloading



overriding

Method overloading:-

When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the no. of arguments (or) a change in the type of arguments.

eg:-

```
Class M1
 {
  Public int sum (int x, int y).
   {
    return (x+y);
   }
 Public int sum (int x, int y, int z)
  {
   return(x+y+z);
  }
 Public double (double x, double y)
   {
    return (x+y);
   }
Class Meoverloading
 {
  Public static void main (String[] args)
    {
      M1 m = new M1;
   System.out.println (m.sum(10,20));
   System.out.println (m.sum(10,20,30));
   System.out.println (m.Sum(10.5, 20.5));
 }
```

o/p:-

```
30
60
31.0
```

Runtime Polymorphism in Java ; (or) Pure Polymorphism :-

It is also known as Dynamic Method dispatch. It is a process in which a function call to the overridden method is resolved at runtime. This type of polymorphism is achieved by Method overriding. Method overridden is occurs when a derived class has a defination for one of the member functions of the base class. That baseclass is said to be overriden.

```
class Bike
 {
 void run()
  {
  System.out.Println("running");
  }
}
class splendor extends Bike.
{
void run()
 {
 System.out.println("running safely with 60KM");
 }
 public static void main(String args[])
  {
  Bike b = new splendor(); // upcasting
   b.run();
  }
 }
```

O/P:-

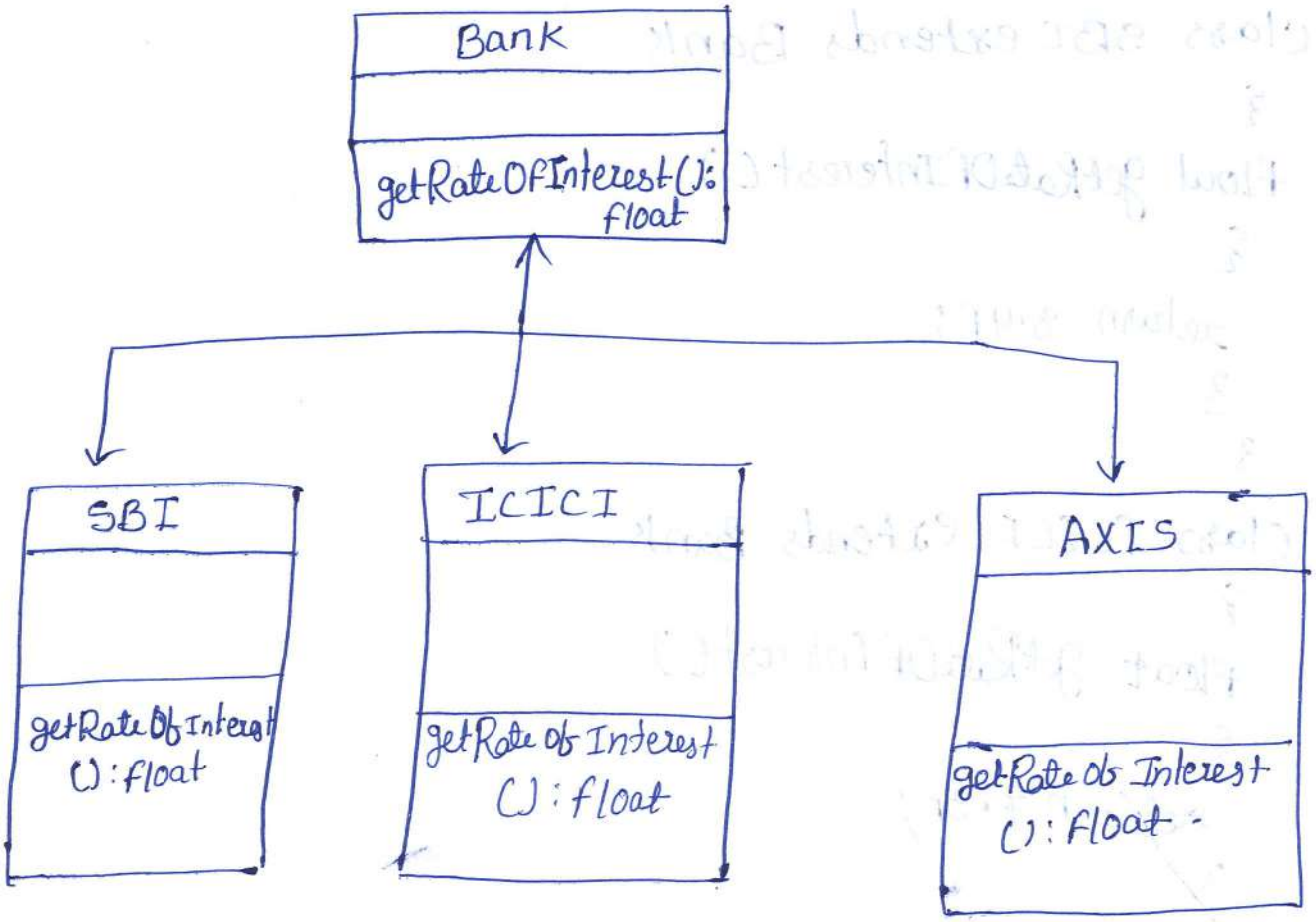
running safely with 60 km

upCasting :-

If the reference variable of Parent class refers to the object of child class, it is known as upcasting.

```
Class A
 {
  }
Class B extends A
  {
  A a = new B();//UpCasting
  }
```

Example for Java Runtime Polymorphism :-



where Bank is a class that Provides a method to get the rate of interest. However the rate of interest may differ according to banks. for example SBI, ICICI, and AXIS banks are prowiding 8.4%, 7.3% and 9.7% rate of interest.

eg:

```
Class Bank {
  float getRateOfInterest()
   {
   return 0;
   }
  Class SBI extends Bank
   {
   float getRateOfInterest()
    {
     return 8.4f;
    }
   }
  Class ICICI extends Bank
   {
   float getRateOfInterest()
   {
    return 7.3f;
   }
  }
```

```
Class AXIS extends Bank
 {
  float getRateOfInterest()
  {
   return 9.7f;
  }

 Class Testpolymoophism
  {
  Public Static void main(Stoing[] args)
   {
    Bank b = new SBI(); //upCasting.
 System.out.println("SBI Rate of Interest:" +b.getRateOfInterest());
  Bank b = new ICICI(); //UPCasting
System.out.println("ICICI Rate of Interest" +b.getRateOfInterest());
  Bank b = new AXIS(); //upCasting
System.out.println("AXIS Rate of Interest" +b.getRateOfInterest());
  }
 }
```

O/P:-

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

Abstract classes:-

An abstract class is a class that created using abstract keyword. a class prefixed with abstract keyword is known as an abstract class.

In java an abstract class may contain abstract methods (methods without implementation) and also non abstract methods (methods with implimentation

eg:-

```
import java.util.*;
abstract class Shape {
    int length, breadth, radius;
    Scanner input = new Scanner(System.in);
    abstract void printArea();
}
class Rectangle extends Shape {
    void printArea() {
    System.out.println("Finding the area of rectangle");
    System.out.println("Enter length & breadth");
        length = input.nextInt();
        breadth = input.nextInt();
System.out.println("The area of Rectangle is:" + length*breadth);
}
```

```
Class Triangle extends Shape {
    Void PointArea() {
    System.out.Println("\n * Finding the Area of Triangle ***");
    System.out.print("Enter Base & Height:");
        length = input.nextInt();
        breadth = input.nextInt();
    System.out.Println("The area of Triangle is:" + (length*breadth)/2);
}
}

Class Circle extends Shape {
    Void Printarea() {
    System.out.Println("In Finding the area of circle");
    System.out.print("Enter radius");
        radius = input.nextInt();
    System.out.Println("The area of circle is" + 3.14f * radius * radius);
}
}

public Class AbstractClass {
  public static void main(String args[]) {
    Rectangle rec = new Rectangle();
        rec.pointArea();
```

```
Triangle tri = new Triangle();
  tri.PrintArea();

Circle Cri = new Circle();
  Cri.PrintArea();
}
```

O/P

```
Finding The area of Rectanagle
Enter Length & Bredth:
2
4
The area of Rectangle is: 8

* Finding The Area of Triangle *
Enter Base & Height: 4 5
The area of triangle: 10
Finding The Area of circle.
Enter Radius: 6
The area of circle is: 113.04
```

Forms of Inheritance :-

The Inheritance Concepts used for the number of purposes in the Java Programming Language one of the main purpose is substitutability. The substitutability means that when a child class acquire properties from its Parent class. The object of the Parent class may be substituted with the child class object.

The substitutability can achieve using inheritance whether using extends (or) implements keywoods.

In the Inheritance we are having 6 Forms.

→ specialization
→ specification
→ Constoction
→ Eextention
→ Limitation
→ Combination.

Specialization:

It is the most ideal foom of inheritance. The subclass is a special case of the Parent class. It holds the principle of substitutability.

Specification :- This is another commonly used foom of inheritance In this foom of inheritance the Parent class just specifies the which methods should be available to the child class but doesn't implement them

The Java Provides Concepts like abstract and interfaces to support this form of inheritance. It holds the principle of substitutability.

Construction:-

This is another form of inheritance where the child class may change the behaviour defined by the parent class (overriding). It doesnot hold the principle of substitutability.

Eextension:-

This is another form of inheritance where the child class may add its new properties. It holds the principle of substitutability.

Limitation:-

This is another form of inheritance where the subclass restricts the inherited behaviour. It doesnot hold principle of substitutability.

Combination:-

This is another form of inheritance where the subclass inherits properties from multiple parent classes. Java doesnot support multiple inheritance type.

Rules for Java Abstract Class:

→ An abstract class must be declared with an abstract keyword.

→ It can have abstract and non-abstract methods.

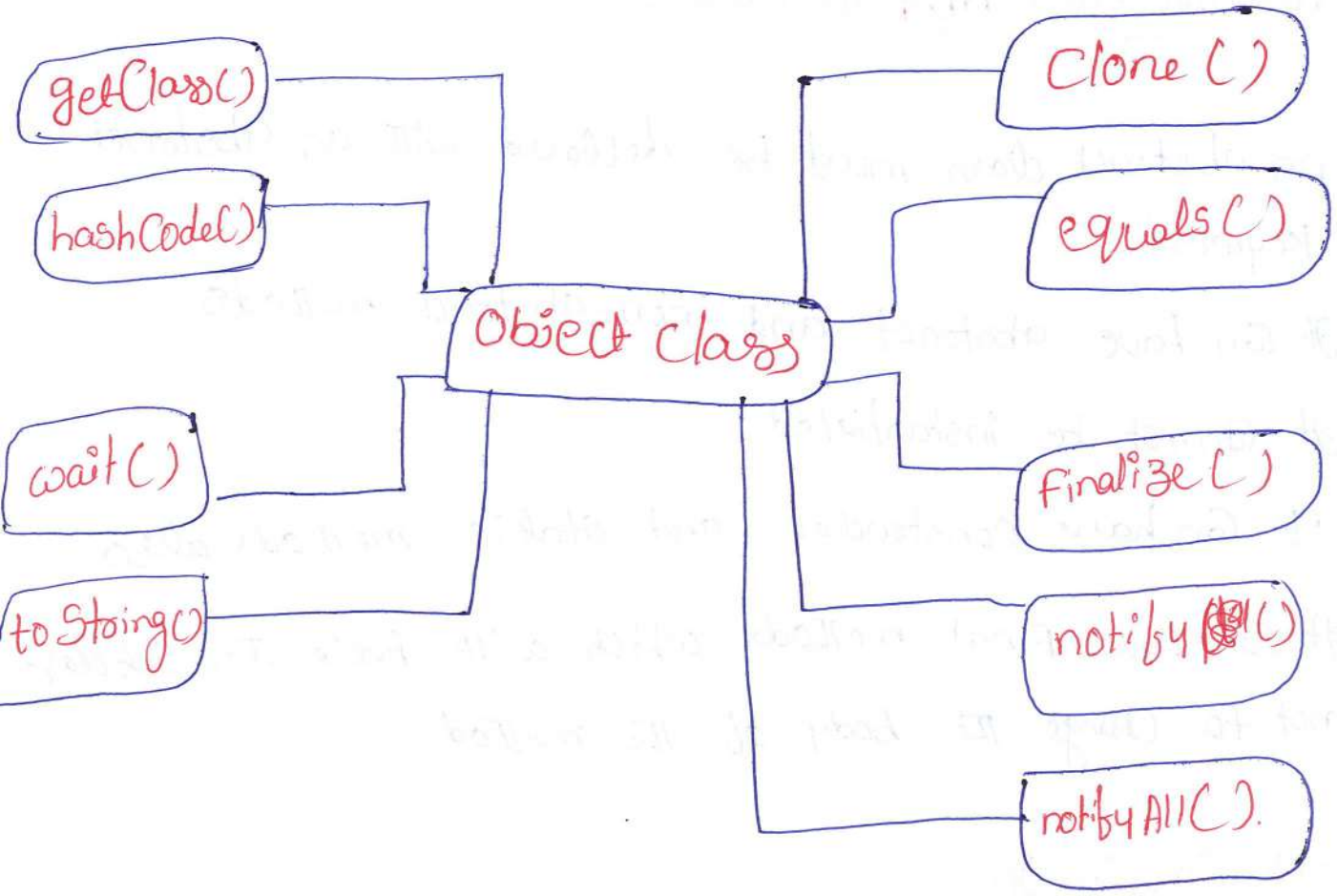
→ It cannot be instantiated

→ It can have constructors and static methods also

→ It can have final methods which will force the subclass not to change the body of the method

Object Class in Java:-

Object class present in java.lang Package. Every class in java is directly(or) indirectly derived from the Object Class. If a class does not extend any other class Then it is a direct child class of Object & if extend another class then it is indirectly derived. Therefore the object class methods are available to all java classes. Hence object class acts as a root of inheritance hierarchy in any java program.



Object class methods :-

(i) toString() :-

toString ~~represents~~ provides String representation of an object and used to convert an object to String.

2. hashCode( ) :-

For every object. JVM Generates a unique number which is hash code. It returns distinct integers for distinct objects.

3. equals(object obj) :

equals(object obj) method is used to compare two objects and its internally uses hash code method to compare objects.

4. getClass( ) :-

getClass() returns the class of an object. It can also be used to get metadata of this class.

5. Finalize( ) :-

This method is called just before an object is garbage collected. it is called by the Garbage Collector on an object when garbage collecter determine that there are no more references to the object.

6. clone( ) :

It returns a new object that is exactly the same as this object. for clone() method refer clone()

The remaining three methods notify( ), notifyall( ), wait( ) are related to Inter-Thread Communication.

eg:-

```
Public class Mainclass {
    Public static void main (Stoing[] args ) {
        Mainclass M = new Mainclass();
        Main class M1 = new Mainclass();
        System.out.Println ("toStoing method "+M.toStoing())
        System.out.Println ("hashCode method"+ M.hashCode())
        System.Out.Println ("equals method "+M.equal(M1));
        System.out.Println ("getClass method"+M.getClass());
    }
}
```

O/P:-

```
tostoing method ;
hashCode method:
equals method :
get class method :
```

Benifits of Inheritance:

→ Inheritance helps in code reuse. The child class may use the code defined in the parent class without rewriting it

→ Inheritance save time and efforts as the main code need not be written again

→ Inheritance provide a clear model structure which is easy to understand.

→ An inheritance leads to less development and maintainence cost

→ with inheritance, we will be able to override the methods of the base class so that the meaningful implementation of Base class method can be designed in the derived class. An inheritance leads to less development and maintainance costs

→ In inheritance base class can decide to keep some data private so that it cannot be altered by the derived class.

Costs of Inheritance:

→ Inheritance decreases the execution speed due to the increased time and effort it takes the program to jump through all the levels of overloaded classes.

→ Inheritance makes the two classes get tightly coupled. This means one can not be used independently of each other

→ The changes made in the parent class will affect the behaviour of child class too

→ The overuse of inheritance makes the program more complex.

Coping with Complexity!-

→ Coping with Complexity in java requires reducing cognitive load throug modularity, modern language features & clean architectural Patters.

→ As your java Project groow in size and Complexity, maintaining, debugging & extending your Code Can become challinging

Here are some strategies to help you cope with complexity in java:-

1. Modularization:- Breaking down the Code into smaller, self contained modules, Classes (or) Packages to manage complexity. Each module should have a specific responsibility and interact with others through well-defined interfaces:

2. Design Patterns:- Using established design Pattern to solve Common architectural & design proble Design Patterns provide proven solution to recurring challenges in software development

3. Encapsulation :-

Restricting access to class members using access modifier (public, private, protected) to hide implementation details and provide a clear API for interacting with the class.

4. Abstraction :

Creating abstract classes & interfaces to define contracts that classes must adhere to making it easier to work with different implementations

5. Documentation :-

Writing comprehensive documentation including code comments and Javadoc, to explain how the code works, its purpose and how to use it

6. Testing :- Implementing unit tests to ensure that individual components of the code function correctly, which helps identify and prevent bugs

7. Code Reviews :- Collaborating with team members to review and provide feedback on code to catch issues & ensure code quality.

8. Version Control :- using version control systems like Git to manage changes, track history & collaborate with others effectively.

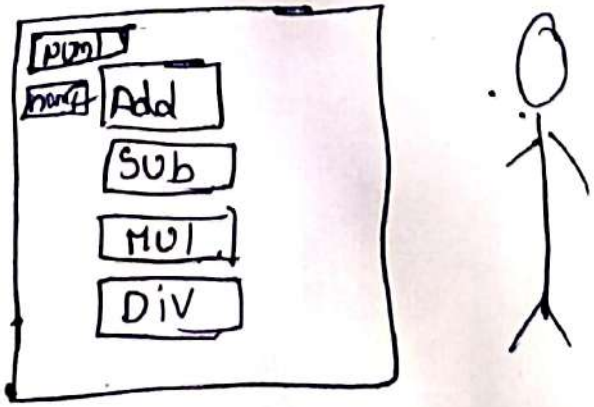
9. Refactoring :- Regularly improving & simplifying the codebase by removing redundancy & improving its structure.

Abstraction Mechanism:

→ An java abstraction is a fundamental concept in OOP that allows you to hide complex implementation details while exposing a simplified & well-defined interface

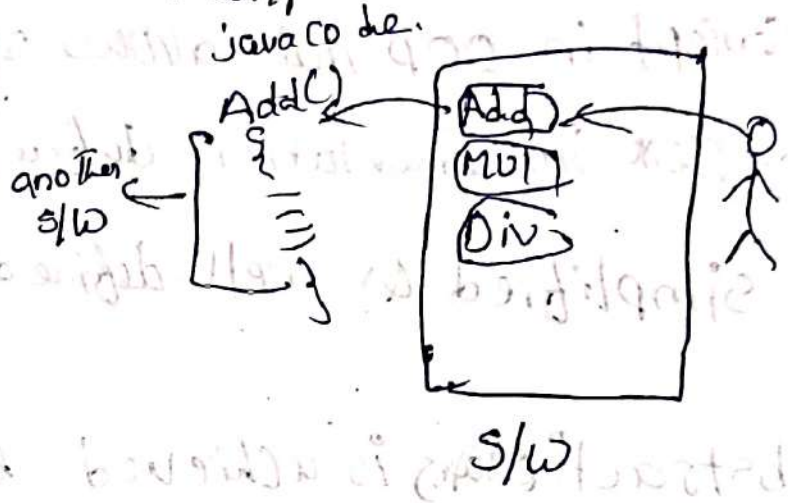
1. Abstraction is achieved by
   ① Interfaces (100%)
   ② Abstract class (0 to 100%)

→ Abstraction is a process of hiding the implementation details from the user only the highlighted set of services provided to the user



Advantage :- 1) Security

2) Enhancement



Abstraction :-

→ abstract class 20% → 80%

→ interface → 100%

Type Conversion and type Casting in java:

Type Conversion (or) implicit type Conversion (or) Automatic type Conversion

It is the process of converting a value from one datatype to another datatype.
This happens when we assign a smaller datatype variable to large datatype variable
No automatic Conversion is supported from numaric type to Char or boolean

lower datatype is converted to higher datatype

byte → short, int, long, float

short → int, long, float

Class Type conversion

```
{
Public Static void main (String[] args)
 {
  int a=10;
  long b=a;
  float c=a;

System.out.println(a+" "+b+" "+c);
 }
}
```

O/P:-

10 10 10.0

Type Casting:-

If the conversion is done explicitly by the programmer (or) user.

Higher datatype value is to be converted to lower datatype.

If we assign a value of higher Datatype to lower datatype then some data will be lost.

long → byte, int, short

float → int, byte, short.

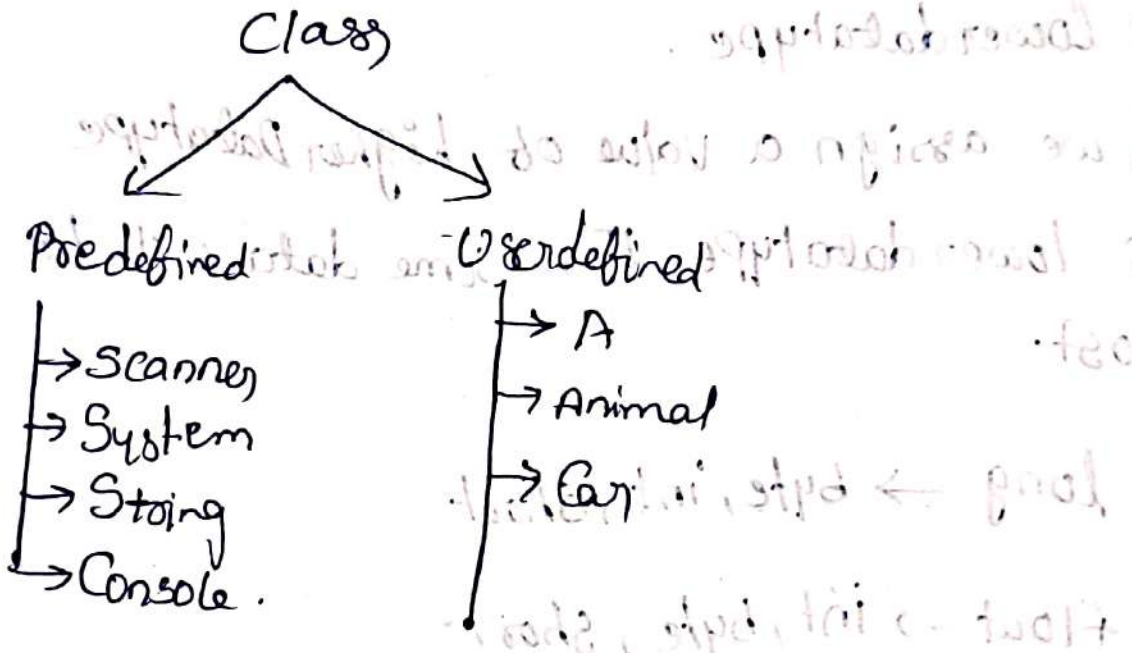
```
Class TypeCasting
{
Public static void main(String[] args)
{
double m = 71.5d;
int a = (int) m;
System.out.println(a);
}
}
```

O/P:-

71

Class:-

Class is a collection of objects and it doesn't take any space on memory.
Class is also called as blueprint / logical entity



User defined class:-

A class which is created by java Programmer is called user defined class

Object:-

Object is an instance of class that executes the class. once the object is created, it takes up space like other variables in memory.

Syntax:- Classname Obj-name = new Class_name();

- Classname: Classname
- Obj-name: object reference
- new: Dynamic Memory Allocation.
- Class_name(): Constructor

eg:-

```
Class Demo
{
int a=10; String b="anitha";
```

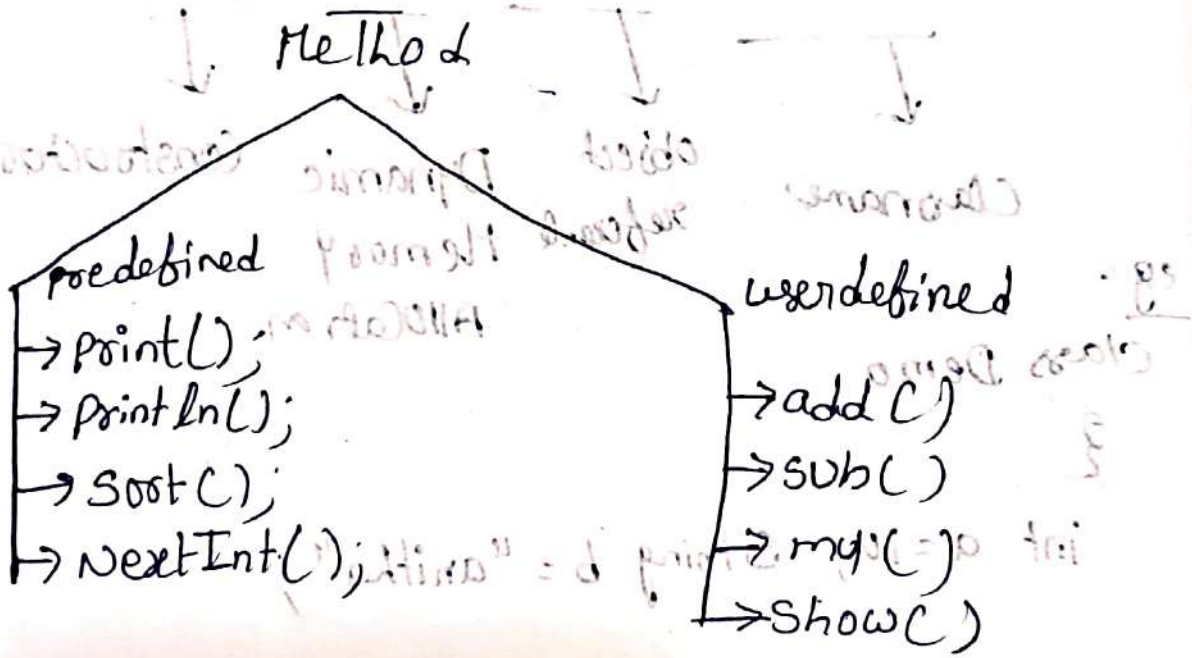
Methods :-

→ Method → calls with in Class

Method is a group/block of Code which take input from the user, Processed it & will give output

Method runs only when it called

Types of Methods :-



Syntax :-

```
returntype function_name(Parameter)
{

statements
}
```

Why we use method?

1→ Decrease Line of code

2→ Readability // Easy to understand by others.

3→ Repetation.

Rules of method declaration

① Defination { = }

2) Calling

```
Class A
{
Public static void main(String[] args)
```

this keyword:-

1) this keyword refers to the current object inside a method or constructor.

2) Whenever the name of instance and local variables both are same then our runtime environment JVM gets confused that which one is local variable & which one is instance variable to avoid this problem we should use this keyword

```
Class A
{
 int a; //Instance variable
 A(int a)
```

```
Class A
{
 int a; // Instance
 A(int a); // Local
  {
   a = a;
  }
 void show()
  {
   System.out.println(a);
  }
 Public static void main (String[] args)
  {
    A r = new A(100);
    r.show();
  }
}
```

3) If is also used when we want to call the default constructor of its own class.

```
Class A
{
 A C)
 {
   System.out.Print("Hello tkroet");
  }
   A(int a)
    {
     this();  → Paranthesis
     System.out.Print(a);
    }
   Public static void main(String[] args)
      A r= new(100);
  }
}
```

O/P:-

Hello tkroet 100.

4) It also called Parameterized Constructor of its own class

```
Class A
{
 A()
 {
  this(10);
 }
 A (int a)
 {
  System.out.Print.(a);
 }
 Public static void main(String[] args)
 {
   A r= new A();
 }
}
```

Constructor :-

↳ Constructor is a special type of method whose name is same as class name.

1) The main purpose of constructor is initialize the object

2) Every java class has a constructor
↓ default constructor
↳ Created by compiler

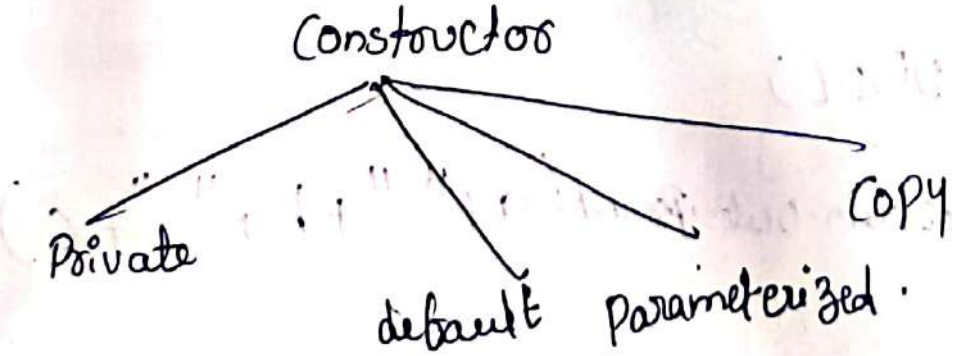
3) A constructor is automatically called at the time of object creation.

4. A constructor never contain any return type including. void

```
Class A
{ int a; String name;
  A()  → Parenthesis.
  {
    a=0; name=null;
  }
  Void show()
  {
    System.out.Print(a+" "+name);
  }
}

Class B
{
Public static void main(String[] args)
  {
    A ref = new A();
    ref.show();
  }
}
```

Types of Constructor:

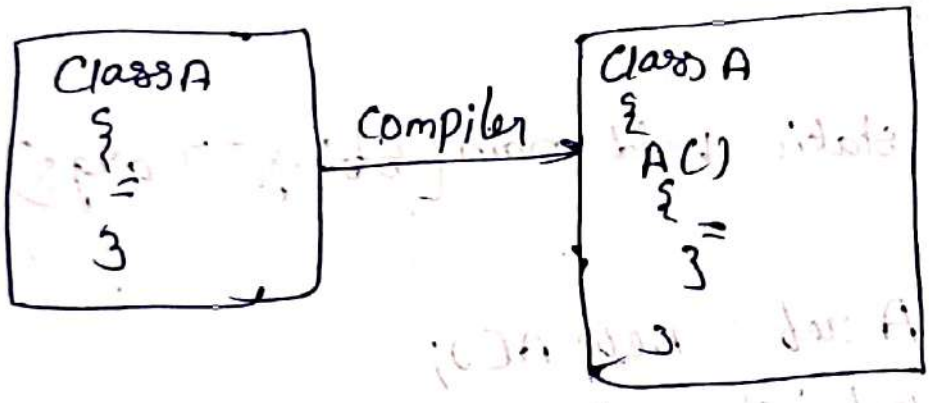


default Constructor:-

→ A constructor which does not have any Parameter is called default Constructor.

```
Class A
{
  A()  → no any Parameter
  {
  }

}
```




```
Class A
{ int a; String b; boolean c;
  A() // default Constructor
  {
    a=100; b="anusha"; c=true;
  }
  void Disp()
    {
      System.out.Print(a+" "+b+" "+c);
    }
}
```

```
Class B
{
  public static void main(String[] args)
  {
    A ref = new A();
    ref.Disp();
  }
}
```

Parameterized Constructor: A constructor through which we can pass one (or) more parameters is called Parameterized constructor.

```
Public Class Dog
  {
    String name;
    Public Dog(String name)
    {
      this.name = name;
System.out.println("A dog named "+name+"is boon");
  }
}
Public class Main{
  Public static void main (String[] args)
    {
      Dog myDog = new Dog ("Rex");
    }
  }
```

o/p:- A dog named Rex is boon

Copy constructor:

```
class A
{  int x; int y;
   A(int a, int b)
   {
      x=a; y=b;
   }
   void show()
   {
      System.out.println(x+" "+y);
   }
}

class B
{
public static void main(String[] args)
   {
      A r= new A(100,200);
      r.show();
   }
}
```

Copy Constructor:-

↳ When ever we Pass object reference to the constructor it is called Copy Constructor.

```
Class Class_name
 {
  Class_name(Obj ref).
   {

   }
```

eg:-

```
Class A
 {
  int a;
  String b;
  A()
  {
   a=10; b="tkrcet"
  }      System.out.println(a+" "+b);
  A(A ref)
   {
   a=ref.a;
   b=ref.b;
   System.out.println(a+" "+b);
  }
```

```
Class B
{
Public static void main(String[] args)
 {
    A r=new A();
    A r2=new A(r);
 }
}
```

Garbage Collection in Java or Java Garbage Collector:

→ In Java when an object is no longer in use or when an object is unreferenced Then Garbage collector (Part of JVM) automatically destroys the object. That's why we can say that Java is excellent in memory management

→ Garbage Collector automatically destroy the object if it is no longer used.

How an object is unreferenced:

↳ 2 ways

1. By nulling the reference.

```
Employee e = new Employee();
e = null;
```

2. By Creating Anonymous object

```
new Employee();
```

In order to call Garbage collector explicitly we uses finalize() and gc().

Finalize(): Protected void finalize()

{

body

}

finalize method is called before each time garbage collector is called.

System.gc(): System class contains a static method called gc() boo requesting JVM to run Garbage Collector.

↓ staticmethod

class

↓

lang.Package.

→ Garbage collection is a process of reclaiming the runtime unused memory automatically.

```
Class Garbage Collection
{
Protected void finalize()
{
System.out.Println("Object memory destroyed");
}
Public static void main(String[] args)
{
Garbage Collection obj = new Garbage Collection();
Obj. = null;
new Garbage Collection(); //Unused object which not having reference Variable
System.gc(); // gt calls finalize
}
}
```

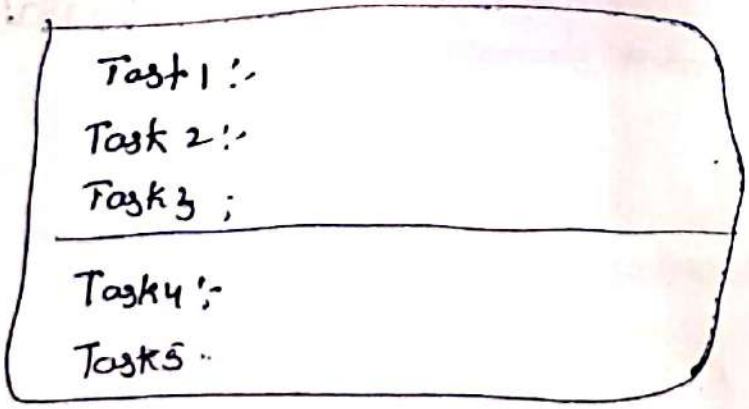
method in object class

default

finalize is a method which is available in Object super Class

The Purpose of finalize method is to release the resources that is allocated by unused object, before removing unused Object by garbage collector.

Garbage collection is the automatic process of eliminating unnecessary Objects from memory



we always free up the space from the memory to avoid Program from terminating abnormally i.e Causing

OutOfMemoryErrors

Making reference NULL

```
Car Car1 = new Car();
Car1 = null;
```

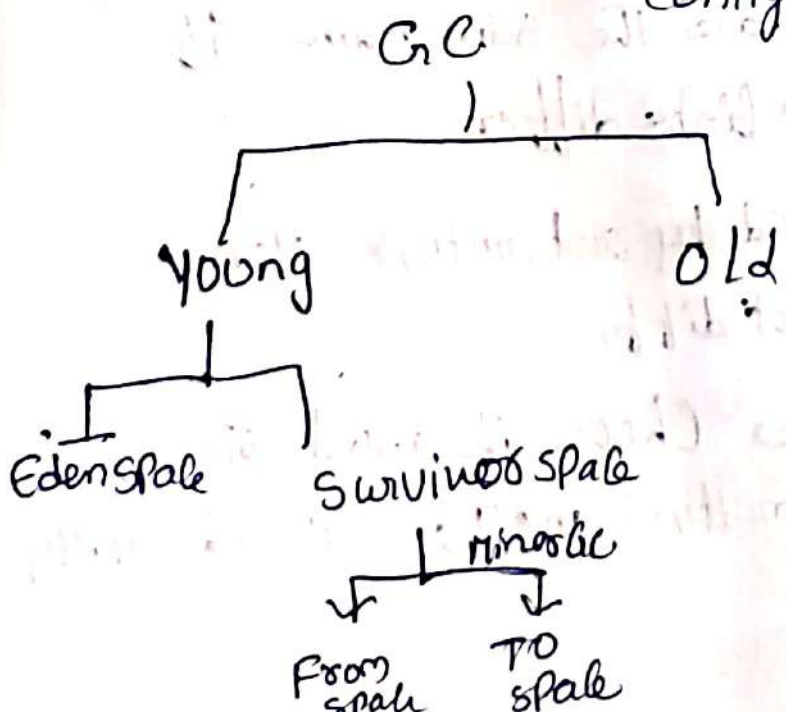
Assigning reference to other

```
Car Car1 = new Car();
Car Car2 = new Car();
Car1 = Car2;
```

Using anonymous Object

```
myMethod(new Car());
```

1. Mark - Marking object alive
2. Sweep - Sweeping dead objects
3. Compact → Arrange objects into the Contiguous part



Overloading methods & constructors:-

Creating two or more methods in the same class sharing a common name but different number of parameters or different types of parameters is called overloading metho

```
Class Addition
  {
    Void sum(int a, int b)
    {
      System.out
```

Method overloading in java allows a class to have multiple methods with the same name but different parameters enabling Compile time Polymorphism.

→ Methods can share the same name if their parameters lists differ

→ Can not overload by return type alone. Parameters must differ

→ The compiler choose the most specific match when multiple methods could apply

```
import java.io.*;
Class Product
  {
 Public int multiply(int a, int b)
    {
    int Prod = a*b;
       return Prod;
  }
 Public int multiply (int a, int b, int c)
  {
  int Prod = a*b*c;
     return Prod;
  }
}
class multiplication
  {
 Public static void main (String[] args)
   {
   Product ob = new Product();
   int Prod1 = ob.multiply(1,2);
System.out.Println ("Product of two integer
                    Values" + prod1);
int prod2 = ob.multiply (1,2,3);
System.out.println ("Product of The Three integer Values"+
                    prod2);
```

Changing the Datatypes of Parameters:

```
Class Product1
  {
Public int prod(int a, int b, int c)
    {
    return a*b*c;
   }

Public double Prod(double a, double b, double c)
    {
     return a*b*c;
    }
  }

Public class tkrcet
    {
Public static void main(String[] args)
  {
   Product1 p = new Product1();

System.out.Println(p.Prod(1,2,3));
System.out.Println(p.Prod(1.0,2.0,3.0));
  }
}
```

o/p:-

6

6.0

Constructor overloading :-

Java supports Constructor overloading which allows a class to have more than one Constructor with different Parameters lists. The appropriate Constructor is selected at Compile time based on the arguments passed during object Creation.

Constructor overloading enables objects to be initialized in multiple ways, improving flexibility and code clarity.

```
Class Box
  {
  double width, height, depth;
  Box (double w, double h, double d)
    {
    width = w;
    height = h;
    depth = d;
  Box ()
    {
    width = height = depth = 0;
    }
```

```
Box (double len)
  {
  width = height = depth = len;
  }

double volume()
   {
  return width * height * depth;
  }

}

Public class Test
{
  public static void main(String[] args)
  {
  Box mybox1 = new Box(10,20,15);
  Box mybox2 = new Box();
  Box myCube = new Box(7);

System.out.println("Volume of mybox1"+
                          mybox1.volume());
System.out.println("Volume of mybox2"+
                          mybox2.volume());
System.out.println("Volume of myCube"+myCube.
                          volume());
  }
}
```

Volume of mybox1 :- 3000.0
Volume of mybox2 :- 0.0
Volume of mycube :- 343.0

Parameter Passing :-

In java we often Pass values to methods so they can perform operations using those values. this is called Parameter Passing. Java supports two types of Parameter Passing

→ Pass-by-value (Primitive Datatypes)
→ Pass-by-reference (Objects & Arrays)

Pass-by-value :-

When we pass a Primitive datatype to a method only a copy of the value is passed. Changes inside the method Do not affect the original value

```
Class Example
{
Static void modifyValue (int num).
 {
   num = num+10;
System.out.Println("Inside method "+num);
}
Public static void main(String[] args)
 {
  int number = 50;
   modifyValue(number);
  System.out.Println("Outside method :"+
                                  number);
 }
}
```

o/p:-

Inside method:60
outside method.

→ Inside the method num. is changed but the original number remains un changed because only a copy was passed.

→ Pass by - reference :- (Passing objects by methods)

when we pass an object to a method a copy of the reference (memory address) is passed. This means changes made inside the method will affect the original object

```
Class Person
 {
  String name;
  Person(String name)
   {
    this.name = name;
   }
  Static void changeName(Person P)
    P.name = "Update name";
 }
 Public static void main (String[] args)
  Person Person1 = new Person("Ram");
  System.out.println("Before:" + Person1.name)
    ChangeName(Person1);
  System.out.println("After:" + Person1.nam
  }
 }
```

o/p:-

Before: Ram

after:- ~~Jhon~~ thrcet

→ The Person Object is Passed to the method and its name property is changed

→ since objects are passed by reference the original object is modified

Passing Arrays to Methods :-

arrays are also Passed by reference. meaning changes inside the method will affect the original Array.

```
Class Example
{
 static void modifyArray(int[] arr){
    arr[0] = 100;
 }
 Public static void main(String[] args).
    int[] numbers = :{1,2,3,4,5}
 System.out.println("Before:"+ numbers[0]);
    modifyArray(numbers);
     System.out.println("After"+ numbers[0]);
}
```

Recursion :-

A function (method) in java can call itself, such calling of function by itself is called Recursion.

Syntax :-

```
return type method_name(Argument ----)
  {
    method_name(Argument ---);
  }
```

```
Class itSelf
 {
 Public static void main(String[] args).
  {
  itself r = new itSelf();
  int a = r.sum(10); // Calling.
  System.out.Println("sum of first 10 Natural numbers:" + a);
  }
 int sum(int b)
  {
   if (b>0)
    {
     return b + sum(b-1);
    }
   else
    {
     return 0;
    }
  }
```

o/p :-

55

nested & inner classes :-

```
Class Animal
 {
 Class Dog
   {
    int legs = 4;
  Void makeSound()
    {
  System.out.Println("bark bark");
    }
  }

Class testclass
  {
Public static void main(String[] args)
 {
 Animal animalObject = new Animal();
 Animal.Dog dogobject = animalObject.new Dog();
System.out.Println("no.of legs of Dog" +dogobject.le
   dogobject.makeSound();
  }
}
```

O/P :-

In java a class with in another class such class is known as nested class

```
class OuterClass
  {

  class Nested class
    {
    }
  }
```

There are two types of nested classes

→ Non-static nested class (inner class)
→ Static Nested class

Non-static Nested Class:

A non-static Nested class is a class with in another class. It has access to members of the enclosing class (OuterClass). It is commonly known as inner class

The inner class exists with in the outerclass you must initiate outerclass first in order to instatiate the inner class

Static Nested class:

we can define a static class inside another class known as nested class

a static nested class cannot access the member variables of outer class. because the static nested class doesn't require you to create an instance of the outer class

```
Class MotherBoard
 {

 Static Class USB
   {
   int usb2 = 2;
   int usb3 = 1;
   int getTotalPorts()
     {
     return usb2 + usb3;
 }
}
Public Class main
   {
   Public static void main(String[] args)
       {
```

```
MotherBoard.USB usb = new MotherBoard.USB();
System.out.println("Total Ports = " + usb.getTotal
                                                Ports());
}
}
```

O/P

Total Ports = 3

Exploring String class :

String is a class available in java.lang Package. using string class we Create Object

1. (a) String s1; // declaration of string.
   s1 = "hello"; // String Literal.

   (b) String s1 = "hello";

2. String s1 = new String("hello");
   Char s[] = {'h', 'e', 'l', 'l', 'o'};
   String s1 = new String(s);

Ways to read input from console:

In Java, there are four different ways for reading input from the user in the command line environment (console).

Input Types:

Method	Description
nextBoolean()	Reads a boolean value from the user.
nextByte()	Reads a byte value from the user.
nextDouble()	Reads a double value from the user.
nextFloat()	Reads a float value from the user.
nextInt()	Reads a int value from the user.
nextLine()	Reads a String value from the user.
nextLong()	Reads a long value from the user.
nextShort()	Reads a short value from the user.

1) Using Buffered Reader Class:

This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.

→ The input is buffered for efficient reading.

→ The wrapping code is hard to remember.

⇒ Program:

```
// Java program to demonstrate BufferReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Test
{
```

Konica

```
psvm(String[] args) throws IOException
throws IOException
{
    //Enter data using BufferReader
    BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    //Reading data using readLine
    String name = reader.readLine();

    //Printing the read line
    System.out.println(name);
}
```

Input: Konica

Output: Konica

Note: To read other types, we use functions like Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

2) Using Scanner Class:

This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however, it is also can be used to read input from the user in the command line.

→ Convenient methods for parsing primitives (nextInt(), nextFloat(), from the tokenized input.

→ Regular expressions can be used to find tokens.

→ The reading methods are not synchronized.

→ Program:

```
// Java program to demonstrate working of Scanner in Java
import java.util.Scanner;

class GetInputFromUser
{
    PSVM (String args[])
    {
        // Using Scanner for Getting Input from User
        Scanner in = new Scanner (System.in);

        String s = in.nextLine();
        S.o.p("You entered string" + s);

        int a = in.nextInt();
        S.o.p("You entered integer" + a);

        float b = in.nextFloat();
        S.o.p("You entered float" + b);
    }
}
```

Input: Welcome
12
3.4

Output: You entered string Welcome
You entered integer 12
You entered float 3.4

Ronica

3) Using Console Class:

It has been becoming a ~~prefeted~~ preferred way for reading user's input from the command line. In addition, it can be used for reading password like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

Advantages:

→ Reading password without echoing the entered characters.
→ Reading methods are synchronized.
→ Format string syntax can be used.
→ Does not work in non-interactive environment (such as in an IDE).

→ **Program:**

```
//Java program to demonstrate working of System.console()
//Note that this program does not work on IDEs as
// System.console() may require console
public class Sample
{
    psvm(String[] args)
    {
        //Using Console to input data from user
        String name = System.console().readLine();
        S.o.p("You entered string" + name);
    }
}
```

Input: Welcome

Output: ~~Note~~ You entered string
Welcome

4) Using Command line argument:

Most used user input for competitive coding. The command-line arguments are stored in the String format. The parseInt method of the Integer class converts string argument into Integer. Similarly, for float and others during execution. The usage of args[] comes into existence in this input form. The passing of information takes place during the program run. The command line is given to args[]. These programs have to be run on cmd.

→ Program:

```
// Program to check for command line arguments
class Hello
{
    PSVM(String[] args)
    {
        // check if length of args array is
        // greater than 0.
        if (args.length > 0)
        {
            S.o.p("The command line arguments are:");
            // iterating the args array and printing
            // the command line arguments
            for(String val: args)
                S.o.p(val);
        }
        else
            S.o.p("No command line" + "arguments found.");
    }
}
```

Command Line Arguments:

```
javac GFG1.java
java Main Hello World
```

~~Please refer this for more faster ways of reading input~~

Output:

```
The command line arguments are:
Hello
World
```

→ Program: (without scanner)

```
Public class UserInput
{
    PSVM(String[] args)
    {
        String name = "Konica Nehal";
        String status = "Im good";
        int age = 21;
        UserInput obj = new UserInput();
        S.o.p("What is your name");
        S.o.p(obj.name);
        S.o.p("Hey," + obj.name + ",How are you?");
        S.o.p(obj.status);
        S.o.p("What is your age?");
        S.o.p(obj.age);
        S.o.p("Thankyou for the information);
    }
}
```

Output:

What is your name
Konica Nehal
Hey Konica Nehal, How are you?
Im good
What is your age?
21
Thankyou for the information

→ Program: (using with scanner)

```
import java.util.Scanner;
public class UserInput
{
    PSVM(String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        S.o.p("What is your name");
        String name = scanner.nextLine();
        S.o.p("Hey %s, How are you? \n", name);
        String status = scanner.nextLine();
        S.o.p("What is your age?");
        int age = scanner.nextInt();
        S.o.p("Thankyou for the information");
        Scanner.close();
    }
}
```

Output:

```
What is your name
Konica Nehal↵
Hey Konica Nehal, How are you?
Im good↵
What is your age?
21↵
Thankyou for the information
```

output:

```
What is your name
Manoj Kumar↵
Hey Manoj Kumar, How are you?
Im very bad↵
What is your age?
52↵
Thankyou for the information
```

Java User Input (Scanner):

The Scanner class is used to get user input, and it is found in the java.util package.

To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class documentation. In our example, we will use the nextLine() method, which is used to read Strings:

→ Program:

```
import java.util.Scanner; //Import the Scanner class
class Main
{
    PSVM(String[] args)
    {
        Scanner myObj = new Scanner(System.in); //create a Scanner object
        S.o.p("Enter username");

        String userName = myObj.nextLine(); //Read user input
        S.o.p("Username is: " + userName); //Output user input
    }
}
```

Output:

```
Enter username
Konica Nehal
Username is: Konica Nehal
```

Program:

```
import java.util.Scanner;
class Main
{ psVM(String[] args)
  {
    Scanner myObj = new Scanner(System.in);
    S.o.p("Enter name, age and salary:");

    //String input
    String name = myObj.nextLine();

    //Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();

    //Output input by user
    S.o.p("Name:" + name);
    S.o.p("Age:" + age);
    S.o.p("Salary:" + salary);
  }
}
```

Output:

```
Enter name, age. and salary:
Konica Nehal↵
21↵
50000↵
Name: Konica Nehal
Age: 21
Salary: 50000
```

Stream based I/O (java.io)

Why we need IO Streams in Java?

The basic need for IO Streams is to

1) Storing and reading data from files
2) Also reading data from the Keyboard

First, let us understand some basic terminology:

Persistence media: The environment that allows us to store data permanently is called persistance Media. We can store data permanently in places, such as

1) File
2) Database
3) Remote Computer (Socket)

Persistence: The process of storing data permanently in persistence media is called persistence.

Persistence logic: The logic that persists data in persistent media is called persistence logic. For example, IOStream based logic, JDBC-based Logic, NetworKing based logic.

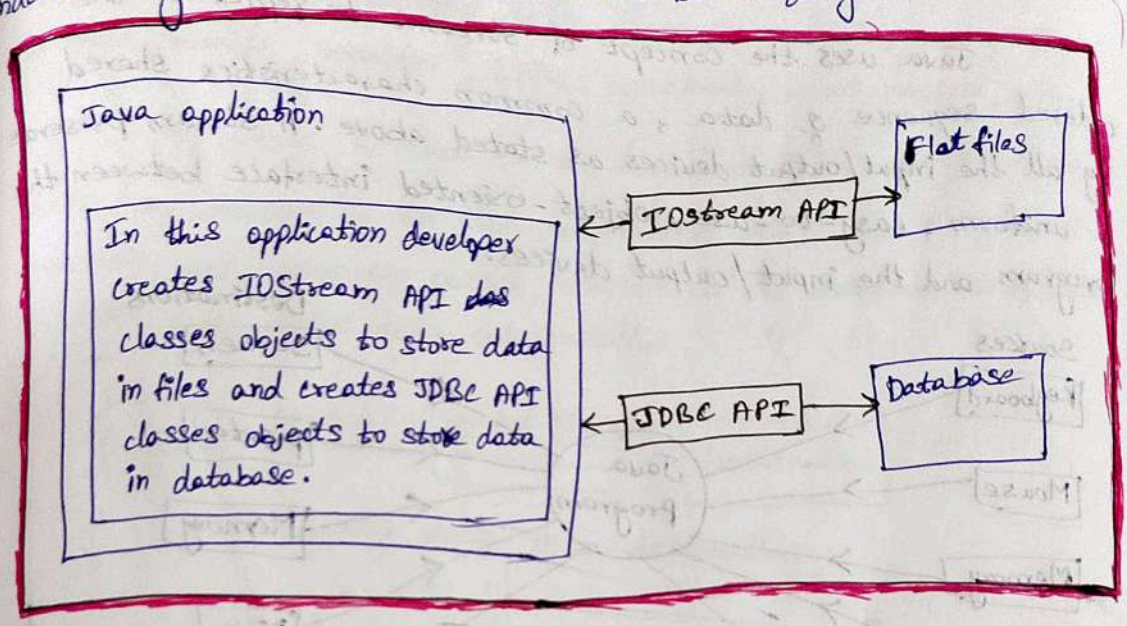
Persistence Technologies: The technology that provides API to develop persistence logic is called persistence technology. Well, Known persistence technologies are IOStream to persist data in files, JDBC, EJB, HIBERNATE to persist data in the database, NetworKing to persist data in Remote Computer.

Where we can store the data or result permanently?

In persistence Media either in files or databases. Storing data in variable and array is temporary. Data will be lost When a local variable goes out of scope or when the program terminates. The programmer uses files or databases for long-term storage of a large amount of data. It is available even after the termination of the program.

To store data in files and Databases SUN has given in-built APIs. We all need to creating a particular class object and calling methods on that object for storing and reading data from that persistence media.

IOStream API is given to store and read data from files. JDBC API is given to store and read data from databases. For better understanding please have a look at the below diagram.



→ We will learn how to store data in files using the IOStream API.
How to store data in the database using the JDBC API.

Note: in either of the approaches we just create objects of the predefined classes and we will call appropriate methods to store and read data from files and databases. So IOStream API and JDBC API are given not for developing business logic rather they are given to develop persistence logic.

How a java application can store and read data from a file?

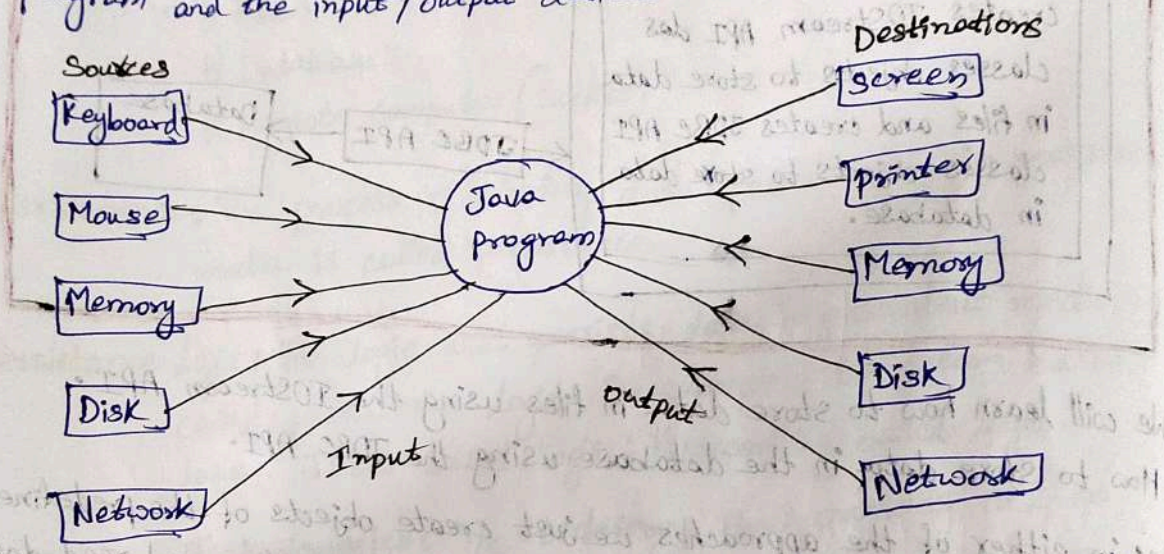
Using Stream object.

What is stream object?

The stream is a logical connection between a java program and a file. To store the data in the persistence media these should be a way to connect to persistence media from java application either physically or logically. Stream provides a logical connection. So we can define the stream as "it is a continuous flow of data between java program and persistence media".

In file processing input refers to the flow of data into a program and output means the flow of data out of a program. Input to a program may come from the keyboard, the mouse, the memory, the disk, a network, or another program. Similarly, output from a program may go to the screen, the printer, the memory, the disk, a network, or another program.

Java uses the concept of streams to represent the ordered sequence of data, a common characteristics shared by all the input/output devices as stated above. A stream presents a uniform, easy-to-use, object-oriented interface between the program and the input/output devices.

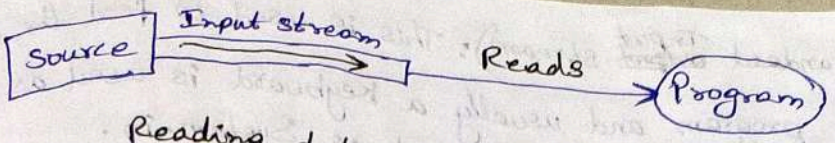


Relationship of Java program with I/O devices

What is Input (I)?

Input means taking the data into our application which is nothing but reading data. Example - command-line arguments, files, networks, gamepads, keyboards, mouse, temperature sensor, webcam, other processes, etc.

An input stream extracts (ie, reads) data from the source (file) and sends it to the program. The program connects and opens an input stream on the data source and reads the data serially.

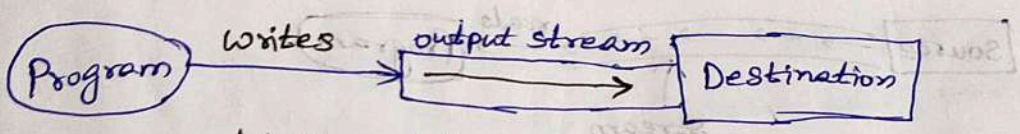


Reading data into a program

What is Output (O)?

Output means giving the data out from our application which is nothing but writing data. Example - files, network, gamepad rumble, monitor, LEDs, speakers, robot motor, etc.

An output stream takes data from the program and sends (ie, writes) it to the destination (file). The program connects and opens an output stream to the destination place of data and writes data out serially.



Writing data to a destination

What are Streams in Java?

Stream meaning is the flow of data from one location to another location. A stream is a data structure facilitating the handling of an information source for sequential access. Programs read from and write to streams, and not the actual source or destination.

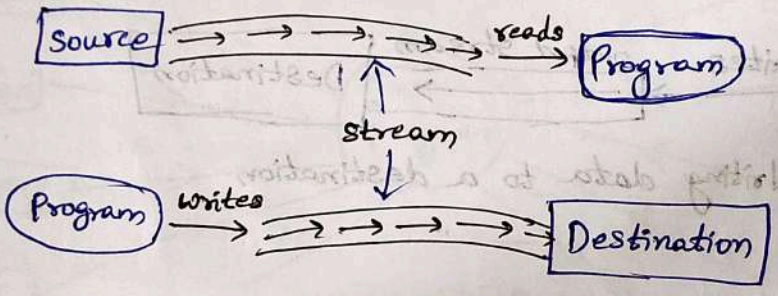
Standard Streams in Java:

In Java, 3 streams are created for us automatically. All these streams are attached to the console.

1) System.in (Standard Input stream): This is used to feed the data to the user's program and usually a keyboard is used as a standard input stream and represented as System.in.

2) System.out (Standard output stream): This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.

3) System.err (Standard error stream): This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.



IO Stream in Java:

IO streams are mainly used to transfer the data from one location to another location. IO streams meant for performing reading operation from or writing operation onto any IO Devices.

Types of IO stream in Java:

Generally, Streams are divided into two types based on the data flow direction, such as

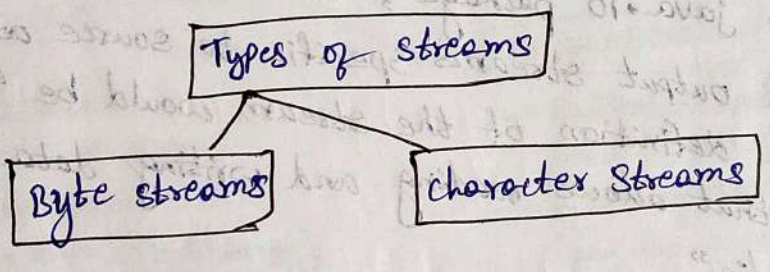
1) Input Stream
2) Output Stream

The stream that allows the data to come into the java application from the persistent media is called an input stream. The stream that allows the data to send out from the java application to be stored into the persistent media is called an output stream.

Basically, InputStreams are used to read data from persistent media, and Output Streams are used to write or store data in a persistent media from the java application. But according to the types of data passed through streams, java streams are divided into two types. They are

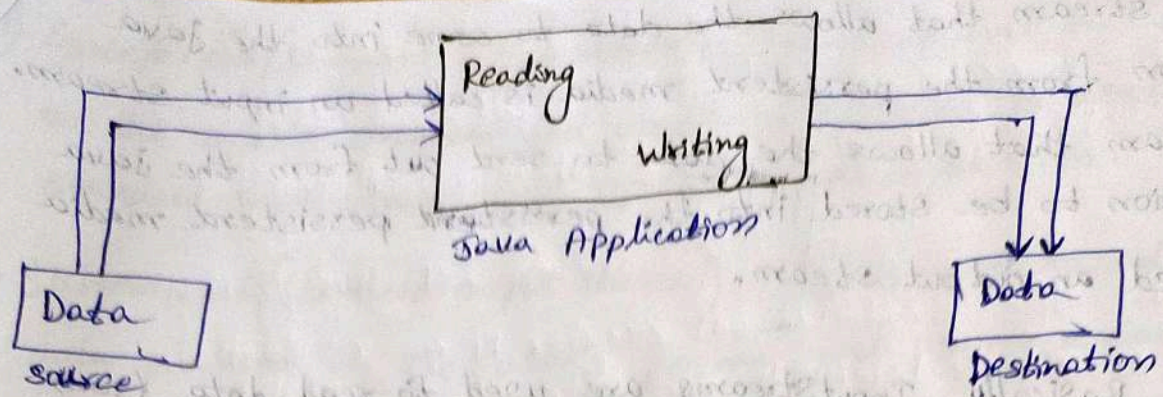
1) Binary Stream
2) Character Stream

The stream which reads and writes the data in the form of bytes is called Binary Streams. The Stream which reads and writes the data in the form of the character is called a Character stream.



What are the different sources and destinations to read and write data?

The data can be read from different sources and also data can be written into different destinations. From sources, the java application reads data and it writes data to the destinations.



The sources can be a Keyboard, mouse, file, Database, socket (computer), object, array, string, etc.
The Destinations can be monitor, File, database, Socket (computer), Object, array, string, etc.

How can we create a Stream in java Specific to source and destination?

In the java.io package, we have several classes to create input and output streams specific to source and destination. So the technical definition of the stream would be "Stream is a java Object that allows reading and writing data to persistence media".

Byte Streams

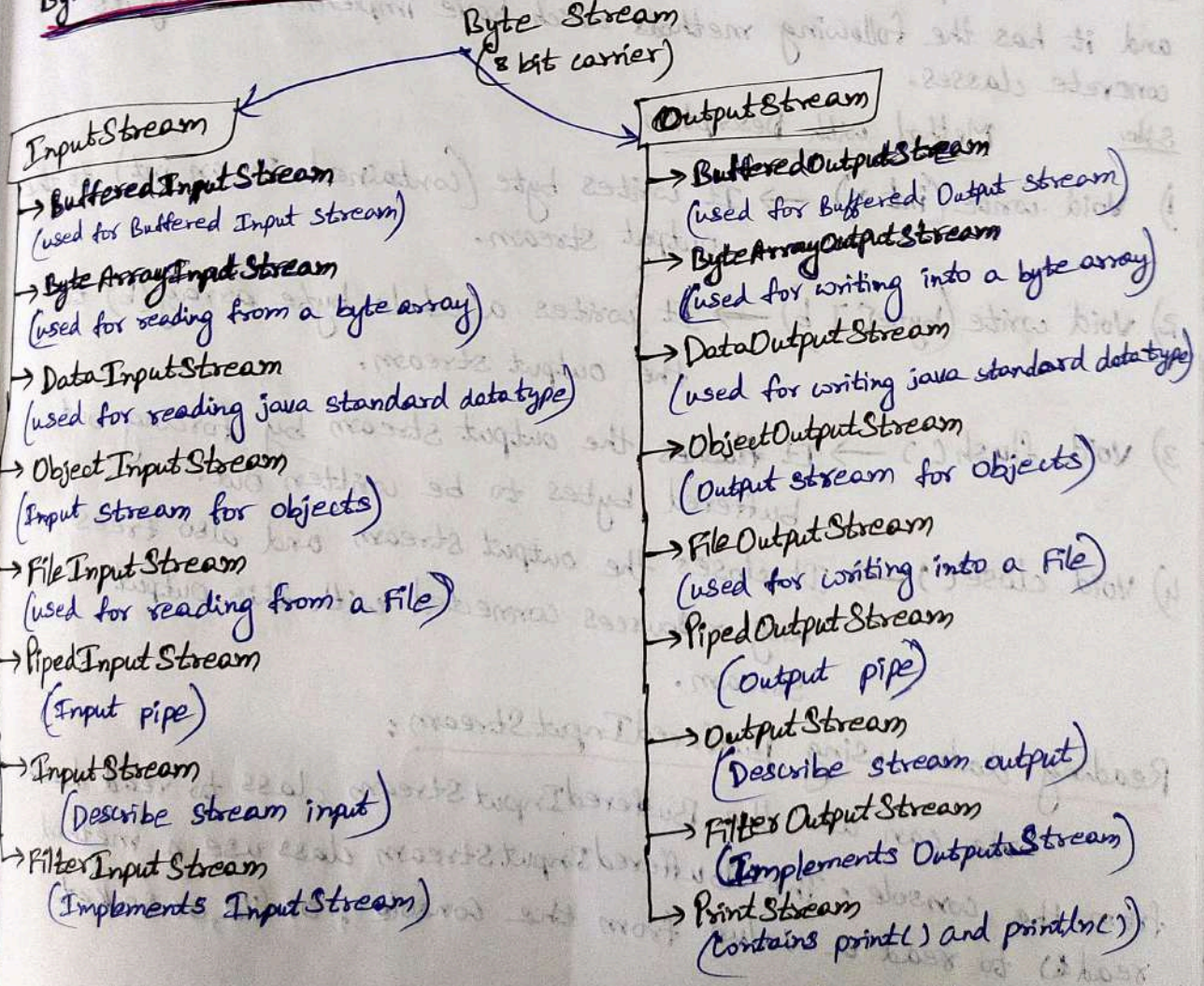
In java, the byte stream is an 8 bit carrier. The byte stream in java allows us to transmit 8 bits of data.

In java 1.0 version all IO operations were byte oriented, there was no other stream (character stream).

The java byte stream is defined by two abstract classes, InputStream and OutputStream. The InputStream class used for byte stream based input operations, and the OutputStream class used for byte stream based output operations.

The InputStream and OutputStream classes have several concreate classes to perform various IO operations based on the byte stream.

Byte stream operations:



read() and write() both are key methods of byte stream.

InputStream class:

The InputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1)	int available() → It returns the number of bytes that can be read from the input stream.
2)	int read() → It reads the next byte from the input stream.
3)	int read(byte[] b) → It reads a chunk of bytes from the input stream and store them in its byte array, b.
4)	Void close() → It closes the input stream and also frees any resources connected with this input stream.

OutputStream class:

The OutputStream class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No.	Method with Description
1)	Void write(int n) → It writes byte (contained in an int) to the output stream.
2)	Void write(byte[] b) → It writes a whole byte array (b) to the output stream.
3)	Void flush() → It flushes the output stream by forcing out buffered bytes to be written out.
4)	Void close() → It closes the output stream and also frees any resources connected with this output stream.

Reading data using BufferedInputStream:

We can use the BufferedInputStream class to read data from the console. The BufferedInputStream class use a method read() to read a value from the console, or file, or socket.

→ Program: Reading from console

```
import java.io.*;
public class ReadingDemo
{
    PSVM(String[] args) throws IOException
    {
        BufferedInputStream readit = new
                BufferedInputStream(System.in);
        try
        {
            S.o.p("Enter any character:");
            char c = (char)readit.read();
            S.o.p("You have entered" + c + "");
        }
        Catch(Exception e)
        {
            S.o.p(e);
        }
        finally
        {
            readit.close();
        }
    }
}
```

Output: (console)

Enter any character: G↵

You have entered 'G'

→ **Program:** Reading from a file

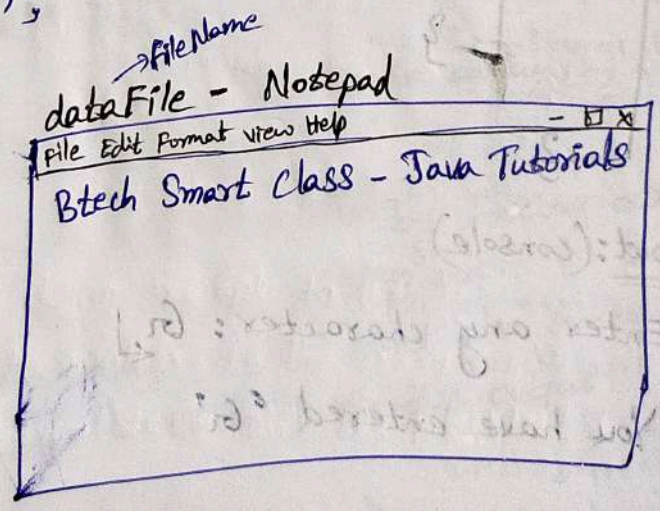
```
import java.io.*;
public class ReadingDemo
{
    PSVM(String[] args) throws IOException
    {
        FileInputStream fileInfo = new
                FileInputStream(new File("C:\\Raja\\dataFile.txt"));

        BufferedInputStream input = new
                BufferedInputStream(fileInfo);

        try
        {
            char c = (char)input.read();
            S.o.p("Data read from a file-" + c + ",");
        }

        Catch(Exception e)
        {
            S.o.p(e);
        }

        finally
        {
            input.close();
        }
    }
}
```



Output: (Console)

Data read from a file - "B"

Writing data using BufferedOutputStream:

We can use the BufferedOutputStream class to write data into the console, file, socket. The BufferedOutputStream class use a method write() to write data.

→ **Program:** Writing data into a file

```
import java.io.*;
public class WritingDemo
{
    PSVM(String[] args) throws IOException
    {
        String data = "Java tutorials by BTech Smart Class";
        BufferedOutputStream out = null;

        try
        {
            FileOutputStream fileInfo = new
                FileOutputStream(new File("C:\\Raja\\dataFile.txt"));
            out = new BufferedOutputStream(fileInfo);
            out.write(data.getBytes());
            S.o.p("Writing data into a file is success!");
        }
        catch (Exception e)
        {
            S.o.p(e);
        }
        finally
        {
            out.close();
        }
    }
}
```

→ FileName

dataFile - Notepad

File Edit Format View Help

Java tutorials by BTech Smart Class

Output: (console)

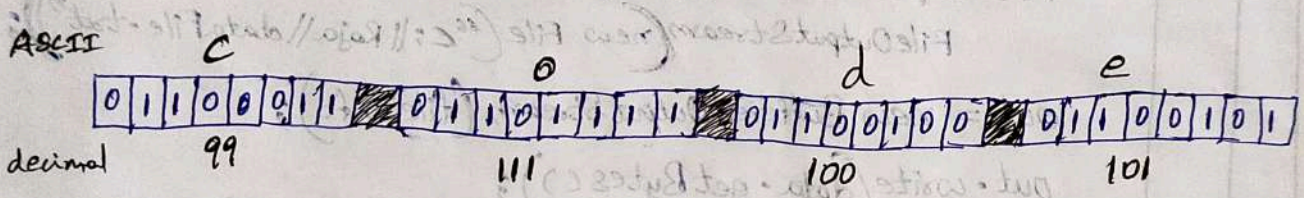
Writing data into a file is success!

Byte Stream:

The data can be interpreted as simple 1's and 0's amounting to the binary value of 10%. However, if this were taken as a smaller segment within an indeterminately-long bitstream - even that interpretation wouldn't be possible.

If one chooses to interpret data as series of 8-bit bytes, many more interpretations based on this minimal structuring are possible. For example, a byte stream could be interpreted in ASCII such that the above value would represent the lowercase character f. Below, is an illustration of how a byte stream representing textual data might appear.

1 byte = 8 bits



This illustrate depicts a 4-byte stream representing the word "code".

Note: In the illustration above, the solid black boxes separating 8-bit groups are purely for visualization purpose only. Such a gap in information would not be present in practice.

→ Program:

Let us take a look at an example to use ByteStream to copy the contents of one file to another. In the example, we will create two objects of the FileInputStream and the FileOutputStream classes.

The source and the destination files names are given as parameters to the FileInputStream and the FileOutputStream classes respectively. Then the content of the source file will be copied to the destination file.

```
import java.io.*;
public class ByteStreamExample
{
    PSVM(String[] args) throws IOException
    {
        FileInputStream sourceStream = null;
        FileOutputStream targetStream = null;
        try
        {
            sourceStream = new FileInputStream("source.txt");
            targetStream = new FileOutputStream("destination.txt");

            // Reading source file using read method
            // and write to file byte to byte using write method
            int temp;
            while ((temp = sourceStream.read()) != -1)
            {
                targetStream.write((byte)temp);
            }
        }
        finally
        {
            if (sourceStream != null)
            {
                sourceStream.close();
            }
            if (targetStream != null)
            {
                targetStream.close();
            }
        }
    }
}
```

Explanation: Let us assume that we have already created a file named source.txt which has the following content -

```
1010000100011111111
```

After we execute the above program, it will create a file called destination.txt in the same directory as the program file which has the same content as of source.txt. If a file with the same name already exists, an exception of FileAlreadyExistsException is thrown.

~~In the code above, a file nam~~

Character Streams

In the code, a file named source.txt is opened and read byte by byte and written to a file object (via byte stream) until the value -1 is reached (java for no more bytes). The end result being that source.txt is copied to destination.txt.

Note: Byte streams should always be closed after use to ensure no memory leaks. A more complete version of the code, demonstrating proper close() methods calls, can be found on GitHub.

Character Stream

In java, when the IO stream manages 16-bit Unicode characters, it is called a character stream. The Unicode set is basically a type of character set where each character corresponds to a specific numeric value within the given character set, and every programming language has a character set.

In java, the character stream is a 16 bits carrier. The character stream in java allows us to transmit 16 bits of data.

The character stream was introduced in Java 1.1 version. The character stream

The java character stream is defined by two abstract classes, Reader and Writer. The Reader class used for character stream based input operations, and the Writer class used for character stream based output operations.

The Reader and Writer classes have several concrete classes to perform various IO operations based on the character stream.

char = 2 bytes = 16 bits

Character Stream Operations:

Character Stream
(16 bits carrier - Unicode)

Reader

- BufferedReader
  (used for Buffered Input Stream)
- CharArrayReader
  (used for reading from an array)
- StringReader
  (used for read from a string)
- FileReader
  (used for reading from a File)
- PipedReader
  (Input pipe)
- InputStreamReader
  (translates bytes to character)
- FilterReader
  (filtered reader)
- LineNumberReader
  (used to count lines)

Writer

- BufferedWriter
  (used for Buffered Output Stream)
- CharArrayWriter
  (used for writing into an array)
- StringWriter
  (used for write into a string)
- FileWriter
  (used for writing into a File)
- PipedWriter
  (Output pipe)
- OutputStreamWriter
  (characters to bytes)
- FilterWriter
  (filtered writer)
- PrintStream
  (Contains print() and println())

read() and write() both the key methods of byte stream

Reader class:

The Reader class has defined as an ~~object~~ abstract class, and it has the following methods which have implemented by its concrete classes.

S.No. Methods with Description

1) int read() → It reads the next character from the input stream.

2) int read(char[] cbuffer) → It reads a chunk of characters from the input stream and store them in its byte array, cbuffer.

3) int read(char[] ~~cbuffer~~ cbuf, int off, int len) → It reads characters into a portion of an array.

4) int read(CharBuffer target) → It reads characters into into the specified character buffer.

5) String readLine() → It reads a line of text. A Line is considered to be terminated by any oneof a line feed ('\n'), a carriage return ('\r'), or a carriage returnfollowed immediately by a linefeed.

6) boolean ready() → It tells whether the stream is ready to be read.

7) Void close() → It closes the input stream and also frees any resources connected with this input stream.

Writer class:

The Writer class has defined as an abstract class, and it has the following methods which have implemented by its concrete classes.

S.No	Methods with Description
1)	void flush() → It flushes the output stream by forcing out buffered bytes to be written out.
2)	void write(char[] cbuf) → It writes a whole array (cbuf) to the output stream.
3)	void write(char[] cbuf, int off, int len) → It writes a portion of an array of characters.
4)	void write(String str) → It writes a string.
5)	void write(int c) → It writes single character.
6)	void write(String str, int off, int len) → It writes a portion of a string.
7)	Writer append(char c) → It appends the specified character to the writer.
8)	Writer append(CharSequence csq) → It appends the specified character sequence to the writer.
9)	Writer append(CharSequence csq, int start, int end) → It appends a subsequence of the specified character sequence to the writer.
10)	void close() → It closes the output stream and also frees any resources connected with this output stream.

Reading data using BufferedReader:

We can use the BufferedReader class to read data from the console. The BufferedInputStream class needs InputStreamReader class. The BufferedReader use a method read() to read a value from the console, or file, or socket.

Let's look at an example code to illustrate reading data using BufferedReader.

→ **Program:** Reading from console

```
import java.io.*;
public class ReadingDemo
{
    PSVM(String[] args) throws IOException
    {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader obj = new BufferedReader(isr);

        String name = "";
        S.o.p("Please enter your name:");
        name = obj.readLine();
        S.o.p("Hello," + name + "!");
    }
}
```

Output: (Console)

```
Please enter your name: Konica
Hello, Konica!
```

→ **Program:** Reading from a file

```
import java.io.*;
public class ReadingDemo
{
    PSVM(String[] args) throws IOException
    {
        Reader in = new FileReader();
        try
        {
            char c = (char) input.read();
            S.o.p("Data read from a file - " + c + "");
        }
        catch(Exception e)
        {
            S.o.p(e);
        }
        finally
        {
            input.close();
        }
    }
}
```

FileName → dataFile - Notepad

File Edit Format View Help

Java tutorials by BTech Smart class

output:

```
Data read from a file - 'J'
```

Writing data using FileWriter:

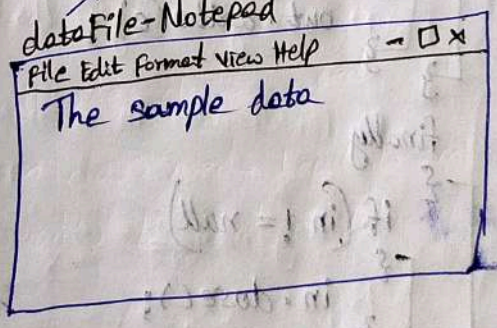
We can use the FileWriter class to write data into the file(). The FileWriter class use a method write() to write data.

Let's look at an example code to illustrate writing data into a file using FileWriter.

→ Program: Writing data into a file

```
import java.io.*
public class WritingDemo
{
    psvm(String[] args) throws IOException
    {
        Writer out = new FileWriter("c:\\Raja\\dataFile.txt");
        String msg = "The sample data";
        try
        {
            out.write(msg);
            S.o.p("Writing done!!!");
        }
        catch(Exception e)
        {
            S.o.p(e);
        }
        finally
        {
            out.close();
        }
    }
}
```

fileName → dataFile-Notepad



Output:

Writing done!!!

Program:

This example deals with the usage of Character Stream to copy the contents of one file to another. In the example, we will create two objects of the FileReader and the FileWriter classes. The source and the destinations files names are given as parameters to the FileReader and the FileWriter classes respectively. Then the content of the source file will be copied to the destination file.

```
import java.io.*;
public class CharacterStreamExample
{
    PSVM(String args[]) throws IOException
    {
        FileReader in = null;
        FileWriter out = null;

        // Reading source file using read method
        // and write to file using write method
        try
        {
            in = new FileReader("source.txt");
            out = new FileWriter("destination.txt");
            int c;
            while((c = in.read()) != -1)
            {
                out.write(c);
            }
        }
        finally
        {
            if(in != null)
            {
                in.close();
            }
            if(out != null)
            {
                out.close();
            }
        }
    }
}
```

Explanation : Let us assume that we have already created a file named source.txt which has the following content -

Hi, we are learning about character stream.

After we execute the above program, it will create a file called destination.txt which has the same content as of source.txt. If a file with the same name already exists, an exception of FileAlreadyExistsException is thrown.

Console IO Operations

Reading Console Input:

In java, there are three ways to read console input. Using the following ways, we can read input data from the console.

1) * Using BufferedReader class
2) * Using Scanner class
3) * Using Console class

1) Reading console input using BufferedReader class in java:

Reading input data using the BufferedReader class is the traditional technique. This way of the reading method is used by wrapping the System.in (standard input stream) is an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.

The BufferedReader class has defined in the java.io package.

→ Program:

```
import java.io.*;
public class ReadingDemo
{
    PSVM(String[] args) throws IOException
    {
        BufferedReader in = new BufferedReader(new
                                InputStreamReader(System.in));
        String name = " ";
        try
        {
            S.O.P("Please enter your name:");
            name = in.readLine();
            System.out.println("Hello," + name + "!");
        }
        catch(Exception e)
        {
            S.O.P(e);
        }
        finally
        {
            in.close();
        }
    }
}
```

output: (console)

```
Please enter your name: Konica↵
Hello, Konica!
```

Reading console input using Scanner class in java:

Reading input data using the Scanner class is the most commonly used method. This way of the reading method is used by wrapping the System.in (Standard input stream) which is wrapped in a Scanner, we can read input from the console.

The Scanner class has defined in the java.util package.

→ Program:

```
import java.util.Scanner;
public class ReadingDemo
{
    PSVM(String[] args)
    {
        Scanner in = new Scanner(System.in);
        String name = "";
        S.o.p(print)("Please enter your name:");
        name = in.nextLine();
        S.o.p("Hello," + name + "!");
    }
}
```

output: (Console)

Please enter your name: Konica↵

Hello, Konica!

3) Reading console input using Console class in java:

Reading input data using the Console class is the most commonly used method. This class was introduced in Java 1.6 version.

The Console class has defined in the java.io package.

→ Program:

```
import java.io.*;
public class ReadingDemo
{
    PSVM(String[] args)
    {
        String name;
        Console con = System.console();
        if (con != null)
        {
            name = con.readLine("Please enter your name:");
            S.o.p("Hello," + name + "!");
        }
        else
        {
            S.o.p("Console not available.");
        }
    }
}
```

Output: (Console)

Please enter your name: Konica↵

Hello, Konica!

Writing Console Output:

In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

1) Using print() and println() methods
2) Using write() method

1) Writing console output using print() and println() methods:

The PrintStream is a built-in class that provides two methods print() and println() to write console output. The print() and println() two methods are the most widely used methods for console output.

Both print() and println() methods are used with System.out stream.

The print() method writes console output in the same line. This method can be used with console output only.

The println() method writes console output in a seperate line (new line). This method can be used with console and also with other output sources.

→ Program:

```
public class WritingDemo
{
    PSVM(String[] args)
    {
        int[] list = new int[5];
        for(int i=0; i<5; i++)
            list[i] = i*10;
        for(int i : list)
            S.o.print(i); // prints in same line
        S.o.println("");
        for(int i:list)
            S.o.println(i); // prints in seperate lines
    }
}
```

(Console) Output:

```
0102030 40
0
10
20
30
40
```

2) Writing console output using write() method:

Alternatively, the PrintStream class provides a method write() to write console output.

The write() method take integer as argument, and writes its ASCII equalent character on to the console, it also accept escape sequences.

→ Program:

```
public class WritingDemo
{
    PSVM(String[] args)
    {
        int[] list = new int[26];
        for(int i=0; i<26; i++)
        {
            list[i] = i+65;
        }
        for(int i: list)
        {
            System.out.write(i);
            System.out.write('\n');
        }
    }
}
```

integers are converted into characters according to the ASCII by using write()

Output: (console)

```
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
```

File Class

The File is a built-in class in Java. In java, the File class has been defined in the java.io package. The File class represents a reference to a file or directory. The File class has various methods to perform operations like creating a file or directory, reading from a file, updating file content, and deleting a file or directory.

The File class in java has the following constructors.

S.No.	Constructor with Description
1)	File (String pathname) → It creates a new File instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.
2)	File (String parent, String child) → It creates a new File instance from a parent abstract pathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
3)	File (File parent, String child) → It creates a new File instance from a parent abstract pathname and a child pathname string. If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.
4)	File (URI uri) → It creates a new File instance by converting the given file: URI into an abstract pathname.

The File class in Java has the following methods:

S.No. Methods with Description

1) String getName() → It returns the name of the file or directory that referenced by the current File object.

2) String getParent() → It returns the pathname of the pathname's parent, or null if the pathname does not name a parent directory.

3) String getPath() → It returns the path of current File.

4) File getParentFile() → It returns the path of the currerd file's parent; or null if it does not exist.

5) String getAbsolutePath() → It returns the current file or directory path from the root.

6) boolean isAbsplute() → It returns true if the current file is absolute, false otherwise.

7) boolean isDirectory() → It returns true, if the current file is a direltory; otherwise returns false.

8) boolean isFile() → It returns true, if the current file is a file; otherwise returns false.

9) boolean exists() → It returns true, if the current file ~~is a file, otherwise return false~~ or directory exist; otherwise returns false.

10) boolean canRead() → It returns true if and only if the file specified exists and can be read by the application; false otherwise.

11) boolean canWrite() → It returns true if and only if the file specified exists and the application is allowed to write to the file; false otherwise.

12) long length() → It returns the length of the current file.

13) long lastModified() → It returns the time that specifies the file was last modified.

14) boolean createNewFile() → It returns true if the named file does not exist and was successfully created; false if the named file already exists.

15) boolean delete() → It deletes the file or directory. And returns true if and only if the file or directory is successfully deleted; false otherwise.

16) void deleteOnExit() → It sends a requests that the file or directory needs be deleted when the virtual machine terminates.

17) boolean mkdir() → It returns true if and only if the directory was created; false otherwise.

18) boolean mkdirs() → It returns true if and only if the directory was created; along with all necessary parent directories; false otherwise.

19) boolean renameTo(File dest) → It renames the current file. And returns true if and only if the renaming succeeded; false otherwise.

20) boolean setLastModified(long time) → It sets the last-modified time of the file or directory. And returns true if and only if the operation succeeded; false otherwise.

21) boolean setReadOnly() → It sets the file permission to only read operations; Returns true if and only if the operation succeeded; false otherwise.

22) String[] list() → It returns an array of strings containing names of all the files and directories in the current directory.

23) String[] list(FilenameFilter filter) → It returns an array of strings containing names of all the files and directories in the current directory that satisfy the specified filter.

24) File[] listFiles() → It returns an array of file references containing names of all the files and directories in the current directory.

25) File[] listFiles(FileFilter filter) → It returns an array of file references containing names of all the files and directories in the current directory that satisfy the specified filter.

26) boolean equals(Object obj) → It returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.

27) int compareTo(File pathname) → It compares two abstract pathnames lexicographically. It returns zero if the argument is equal to this pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.

28) int compareTo(File pathname) → compares this abstract pathname to another object. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.

Program:

```
import java.io.*;
public class FileClassTest
{
    PSVM (String[] args)
    {
        File f = new File("C:\\Raja\\datFile.txt");
        S.o.p("Executable File:" + f.canExecuted());
        S.o.p("Name of the file:" + f.getName());
        S.o.p("Path of the file:" + f.getAbsolutePath());
        S.o.p("Parent name:" + f.getParent());
        S.o.p("Write mode:" + f.canWrite());
        S.o.p("Read mode:" + f.canRead());
        S.o.p("Existance:" + f.exists());
        S.o.p("Last Modified:" + f.lastModified());
        S.o.p("Length:" + f.length());
        // f.createNewFile();
        // f.delete();
        // f.setReadOnly();
    }
}
```

Output: (console)

```
Executable File: false
Name of the file: datFile.txt
Path of the file: C:\Raja\datFile.txt
Parent name: C:\Raja
Write mode: false
Read mode: false
Existance: false
Last Modified: 0
Length: 0
```

Program:

```
import java.io.*;
public class FileDemo
{
    PSVM(String[] args)
    {
        try
        {
            File file = new File("javaFile123.txt");
            if(file.createNewFile())
            {
                S.o.p("New File is created!");
            }
            else
            {
                S.o.p("File already exists.");
            }
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Output: New File is created!

File class

File and Directory:

A file is a named location that can be used to store related information. For example

main.java is a Java file that contains information about the Java program.

A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

Create a Java File Object:

To create an object of File, we need to import the java.io.File package first. Once we import the package, here is how we can create objects of file.

```
//creates an object of File using the path
File file = new File(String pathName);
```

Here, we have created a file object named file. The object can be used to work with files and directories.

Note: In Java, creating a file object does not mean creating a file. Instead, a file object is an abstract representation of the file or directory pathname (Specified in the parenthesis).

Java File Operation Methods:

Operation	Method	Package
To create file	createNewFile()	java.io.File
To read file	read()	java.io.FileReader
To write file	write()	java.io.FileWriter
To delete file	delete()	java.io.File

Java create files:

To create a new file, we can use the createNewFile() method. It returns

- → true if a new file is created
- → false if the file already exists in the specified location

→ Program: Create a new file

```
//importing the File class
import java.io.File;
class Main
{
    PSVM (String[] args)
    {
        //create a file object for the current location
        File file = new File("newFile.txt");
        try
        {
            //trying to create a file based on the object
            boolean value = file.createNewFile();
            if (value)
            {
                S.o.p("The new file is created.");
            }
            else
            {
                S.o.p("The file already exists.");
            }
        }
        catch (Exception e)
        {
            e.getStackTrace();
        }
    }
}
```

In the above example, we have created a file object named file. The file object is linked with the specified file path.

```
File file = new File("newfile.txt");
```

Here, we have used the file object to create the new file with the specified path.

If newFile.txt doesn't exist in the current location, the file is created and this message is shown.

The new file is created.

However, if newFile.txt already exists, we will see this message.

The file already exists.

Java read files:

To read data from the file, we can use subclasses of either InputStream or Reader.

→ **Program:** Read a file using FileReader

Suppose we have a file named input.txt with the following content.

This is a line of text inside the file.

Now let's try to read the file using Java FileReader.

```
// importing the FileReader class
import java.io.FileReader;
class Main
{
    PSVM(String[] args)
    {
        char[] array = new char[100];
        try
        {
            // Creates a reader using the FileReader
            FileReader input = new FileReader("input.txt");

            // Reads characters
            input.read(array);
            S.o.p("Data in the file:");
            S.o.p(array);
            // close the reader
            input.close();
        }
        catch (Exception e)
        {
            e.getStackTrace();
        }
    }
}
```

Output:

```
Data in the file:
This is a line of text inside the file.
```

In the above example, we have used created an object of FileReader named input. It is now linked with the input.txt file.

```
FileReader input = new FileReader("input.txt");
```

To read the data from the input.txt file, we have used the read() method of FileReader.

Java write to files:

To write data to the file, we can use subclasses of either OutputStream or Writer.

→ Program: Write to file using FileWriter

```
// importing the FileWriter class
import java.io.FileWriter;
class Main
{
    PSVM (String args[])
    {
        String data = "This is the data in the output file";
        try
        {
            // Creates a Writer using FileWriter
            FileWriter output = new FileWriter ("output.txt");
            // Writes string to the file
            output.write(data);
            S.o.p("Data is written to the file.");
            // closes the writer
            output.close()
        }
        catch (Exception e)
        {
            e.getStackTrace();
        }
    }
}
```

output: Data is written to the file.

In the above example, we have created a writer using the FileWriter class. The Writer is linked with the FileWriter class. The writer is linked with the output.txt file.

```
FileWriter output = new FileWriter ("output.txt");
```

To write data to the file, we have used the write() method.

Here when we run the program, the output.txt file is filled with the following content.

This is the data in the output file

Java delete files:

We can use the delete() method of the File class to delete the specified file or directory. It returns

→ true if the file is deleted.

→ false if the file does not exist.

Note: We can only delete empty directories.

→ **Program:** Delete a file

```
import java.io.File;
class Main
{
    psvm(String[] args)
    {
        // creates a file object
        File file = new File("file.txt");
        //deletes the file
        boolean value = file.delete();
        if(value)
        {
            S.o.p("The File is deleted.");
        }
        else
        {
            S.o.p("The File is not deleted.");
        }
    }
}
```

Output: The File is deleted.

In the above example, we have created an object of File named file. The file now holds the information about the specified file.

```
File file = new File("file.txt");
```

Here we have used the delete() method to delete the file specified by the object.

Reading and Writing Files:

In java, there multiple ways to read data from a file and to write data to a file. The most commonly used ways are as follows.

1) Using Byte Stream (FileInputStream and FileOutputStream).
2) Using Character Stream (FileReader and FileWriter).

1) File Handling using Byte Stream:

In java, we can use a byte stream to handle files. The byte stream has the following built-in classes to perform various operations on a file.

→ FileInputStream - It is a built-in class in java that allows reading data from a file. This class has implemented based on the byte stream. The FileInputStream class provides a method read() to read data from a file byte to byte.

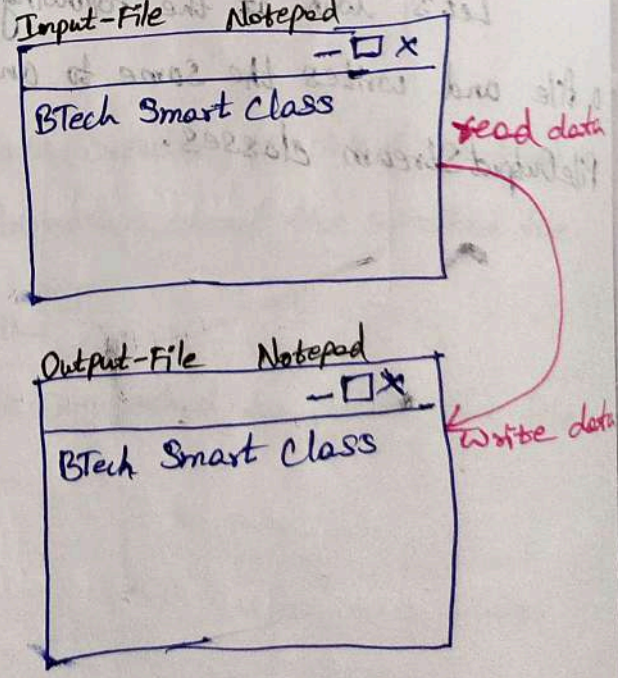
→ FileOutputStream - It is a built-in class in java that allows writing data to a file. This class has implemented based on the byte stream. The FileOutputStream class provides a method write() to write data to a file byte by byte.

Let's look at the following example program that reads data from a file and writes the same to another file using FileInputStream and FileOutputStream classes.

Program:

```
import java.io.*;
public class FileReadingTest
{
    psvm(String args[]) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out = null;

        try
        {
            in = new FileInputStream("C:\\Raja\\Input-File.txt");
            out = new FileOutputStream("C:\\Raja\\Output-File.txt");
            int c;
            while((c = in.read()) != -1)
            {
                out.write(c);
            }
            S.o.p("Reading and Writing has been success!!!");
        }
        catch(Exception e)
        {
            S.o.p(e);
        }
        finally
        {
            if(in != null)
            {
                in.close();
            }
            if(out != null)
            {
                out.close();
            }
        }
    }
}
```



Output: (console).

Reading and Writing has been success!!!

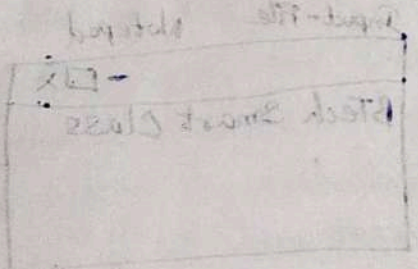
File Handling using Character Stream:

In java, we can use a character stream to handle files. The character stream has the following built-in classes to perform various operations on a file.

→ FileReader - It is a built-in class in java that allows reading data from a file. This class has implemented based on the character stream. The FileReader class provides a method read() to read data from a file character by character.

→ FileWriter - It is a built-in class in java that allows writing data to a file. This class has implemented based on the character stream. The FileWriter class provides a method write() to write data to a file character by character.

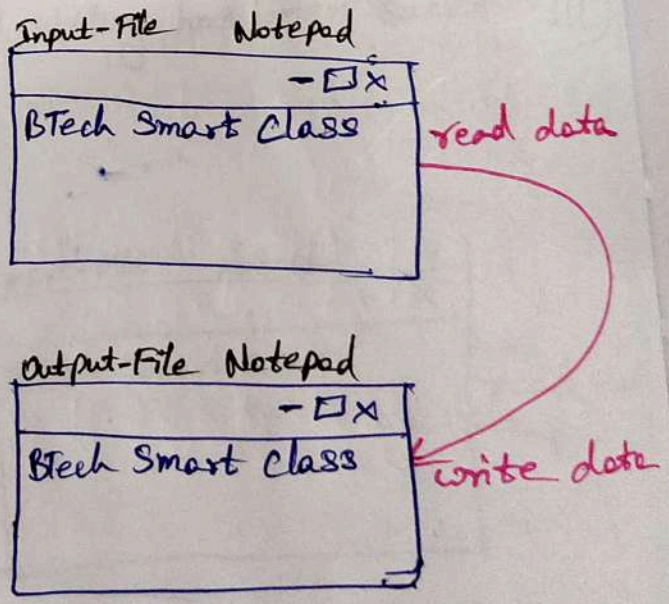
Let's look at the following example program that reads data from a file and writes the same to another file using FileReader and FileWriter classes.



Program:

```
import java.io.*;
public class FileIO
{
    PSVM (String args[]) throws IOException
    {
        FileReader in = null;
        FileWriter out = null;
        try
        {
            in = new FileReader ("C:\\Raja\\Input-File.txt");
            out = new FileWriter ("C:\\Raja\\Output-File.txt");

            int c;
            while ((c = in.read()) != -1)
            {
                out.write(c);
            }
            S.o.p ("Reading and Writing in a file is done!!!");
        }
        catch(Exception e)
        {
            S.o.p(e);
        }
        finally
        {
            if (in != null)
            {
                in.close();
            }
            if (out != null)
            {
                out.close();
            }
        }
    }
}
```



Output: (console)

Reading and Writing in a file is done!!!

Random

Overview:

Rando
of data
write d
and ma

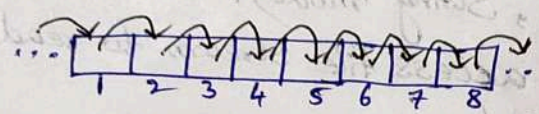
Wha

Random Access File :

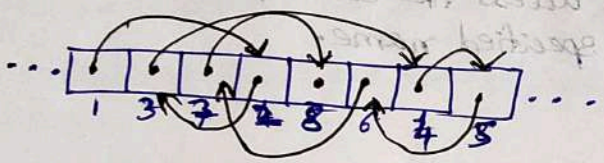
overview:

RandomAccessFile is a Java class that allows the reading and writing of data to any location in the file respectively. It is used to read and write data simultaneously ie., we can read the file, write to a file, and move the file pointer to another location in the same program.

Sequential access :



Random access



What is Random Access File :

RandomAccessFile in java is a class that lets the user read and write to a file at the same time. It extends the Object class and implements DataInput and DataOutput interfaces. These interfaces facilitate the conversion of primitive type data to a sequence of bytes and bytes to specified type of data. It is used to read and write data to a file simultaneously. The file acts as a large array of bytes stored in the file system.

Methods of this class usually throw EOFException ie, End of File Exception" if the end of the file is reached before the desired number of bytes are read. It is a type of IOException.

IOException is thrown if any error occurs other than EOFException like if the byte cannot be read or written.

File Pointer in RandomAccessFile Class:

A file behaves like a large array of bytes, thus we use a cursor or index for the array called the file pointer. Data is read byte by byte starting at the file pointer and the write operation is performed starting at the file pointer too. File pointer can be moved or retrieved using seek() and getFilePointer() methods.

Constructors of RandomAccessFile Class:

1) RandomAccessFile (File fileObj, String mode):
   It creates a random access file stream to read, write, or execute the specified File Object.
2) RandomAccessFile (String fileName, String mode):
   It creates a random access file stream to read, write, or execute the file with the specified name.

Access Modes:

1) r - Creates the file with read mode; calling write methods will result in an IOException. This mode used to read the file but not to write in the file.
2) w - This mode can be used only to write data into it.
3) rwd - Creates the file with read and write mode - Synchronously. All updates to file content is written to the disk synchronously.
4) rws - Creates the file with read and write mode - Synchronously. All updates to file content or meta data is written to the disk synchronously.
5) rw - Creates the file with read and write mode. ~~asynchronously~~

RandomAccessFile methods:

1) int read() - It reads byte of data from a file. The byte is returned as an integer in the range 0-255.
2) int read(byte[] b) - It reads byte of data from file upto b.length, -1 if end of file is reached.
3) int read(byte[] b, int offset, int len) - It reads bytes initialising from offset position upto b.length from the buffer.
4) boolean readBoolean() - It reads a boolean value from the file.
5) byte readByte() - It reads signed eight-bit value from file.

6) char
7) double
8) float
9) long
10) int
11) void
12) voi
13) St
14) v
15)
16)
17)
18)

6) char readChar() - It reads a character value from file.

7) double readDouble() - It reads a double value from file

8) float readFloat() - It reads a float value from file.

9) long readLong() - It reads a long value from file.

10) int readInt() - It reads a integer value from file.

11) void readFully(byte[] b) - It reads bytes initialising from offset position upto b.length from the buffer.

12) void readFully(byte[] b, int offset, int len) - It reads bytes initialising from offset position upto b.length from the buffer.

13) String readUTF() - t reads in a string from the file.

14) void seek(long pos) - It sets the file-pointer (cursor) measured from the beginning of the file, at which the next read or write occurs.

15) long length() - It returns the length of the file.

16) void write(int b) - It writes the specified byte to the file from the current cursor position.

17) void writeFloat(float v) - It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.

18) void writeDouble(double v) - It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

o/p :-

Program: (myFile.java)

```
import java.io.IOException;
import java.io.RandomAccessFile;

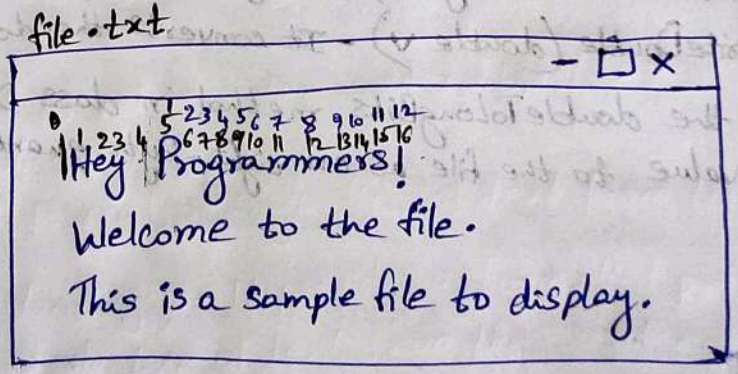
public class myFile
{
    PSVM(String[] args) throws IOException
    {
        //object of RandomAccessFile
        RandomAccessFile raf = new RandomAccessFile("c:\\Raja\\file.txt", "r");

        //moving file pointer
        raf.seek(4);

        byte[] bytes = new byte[12];
        //storing file data
        raf.read(bytes);
        raf.close();
        //printing file data
        S.o.p(new String(bytes));
    }
}
```

Output:

Programmers!



In the example above, we are creating an object of the RandomAccessFile class. It takes two parameters the file along with its location and the mode in which we wish to open the file. The mode can read, write or execute. We are moving the pointer 4 bytes ahead using the seek() method. We read 12 bytes of data from the file pointer into bytes and print it in the output by converting it into the string.

The Console class:

The Java Console class is be used to get input from console. It Provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally.

Let's see a simple example to read text from console.

```
String text = System.console().readLine();
System.out.println("Text is: " + text);
```

In java, most the input functionalities of Console class available through Sytem.in, and the output functionalities available through System.out.

Java Console class declaration:

Let's see the declaration for java.io.Console class:

public final class Console extends Object implements Flushable

Java Console class methods:

1) Reader reader() - It is used to retrieve the reader object associated with the console.
2) String readline() - It is used to read a single line of text from the console.
3) String readline(String fmt, Object... args) - It provides a formatted prompt then reads the single line of text from the console.
4) char[] readPassword() - It is used to read password that is not being displayed on the console.
5) char[] readPassword(String fmt, Object... args) - It provides a formatted prompt then reads the password that is not being displayed on the console.
6) Console printf(String format, Object... args) - It is used to write a string to the console output stream.
7) PrintWriter writer() - It is used to retrieve the PrintWriter object associated with the console.
8) void flush() - It is used to flushes the console.

→ Program:

```
import java.io.Console;
class ReadingStringTest
{
    PSVM(String args[])
    {
        Console c = System.console();
        S.o.p("Enter your name:");
        String n = c.readline();
        S.o.p("Welcome" + n);
    }
}
```

output:

Enter your name: Konica Nehal ↵
Welcome Konica Nehal

→ ~~Example~~ Program: (Read password)

```
import java.io.Console;
class ReadPasswordTest
{
    PSVM(String args[])
    {
        Console c = System.console();
        S.o.p("Enter password:");
        char[] ch = c.readPassword();
        String pass = String.valueOf(ch); // converting char array into String
        S.o.p("Password is:" + pass);
    }
}
```

output:

Enter password: ↵

Password is: 123

Access Modifiers: (Access specifiers)

→ Access Modifiers are used to scope the variables, methods, and other members of a class. It is used to restrict access to different elements of a class.

→ These specifiers are actually keywords in Java.

→ Access modifiers sets the visibility of a variable, constructor, class or a method. There are two types of modifiers in java; access modifers and non-access modifiers in java.

Types of access specifiers:

1) Private
2) Default
3) Protected
4) Default

Note: To better understand access specifiers, we need to have an understanding of packages.

1) Private:

→ Variable or method declared as "private" is only visible within the class it is declared. It is not visible anywhere else including in subclasses of its class.

→ Private access modifiers are restrictive one. Because it restricts the access of variables or methods outside the package.

Note: class and interface cannot be declared as private.

→ The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.

* Any other class of the same package will not be able to access these members.

* Top-level classes or interfaces can not be declared as private because
  → private means "only visible within the enclosing class".
  → protected means "only visible within the enclosing class and any subclasses"

Hence these modifiers in terms of applications to classes, apply only to nested classes and not on top-level classes.

→ Program: We have created two classes A and Simple. A class contains private data members and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
    private int data = 40;
    private void msg()
    {
        System.out.println("Hello Java");
    }
}

public class Simple
{
    PSVM(String args[])
    {
        A obj = new A();
        S.o.p(obj.data); // Compile Time Error
        obj.msg(); // Compile Time Error
    }
}
```

→ Program: (Role of Private Constructor) If you make any class constructor private, you cannot create the instance of that class from outside the class.

```
class A
{
    private A() // private constructor
    {
    }
    Void class Simple
    void msg() // method
    {
        S.o.p("Hello java");
    }
}

public class Simple
{
    PSVM(String args[])
    {
        A obj = new A(); // Compile Time Error
    }
}
```

Note: A class cannot be private or protected except nested class.

→ **Program:** We will create two classes A and B within the same package P1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
Package p1;

class A
{
    private void display()
    {
        S.o.p("Hello");
    }
}

class B
{
    PSVM(String args[])
    {
        A obj = new A();
        // Trying to access private method of another class
        obj.display();
    }
}
```

output:

error: display() has private access in A.
obj.display();

→ **Program:**

Consider a package pkg1 which contains the ~~cat~~ class below.

```
package pkg1;
public class Calculate
{
    private int sum(int a, int b)
    {
        return a+b;
    }
    public int additionOfNums(int x, int y)
    {
        return sum(x, y);
    }
}
```

Consider a package pkg2 which contains the class below. It imports the class Calculate from package pkg1.

```
package pkg2;
import pkg1.Calculate;
public class Program
{
    PSVM (String args[])
    {
        int A = 76;
        int B = 14;
        // This will generate an error
        // int sum = cal.sum(A,B);

        // This is work
        Calculate cal = new Calculate();
        int sum = cal.additionOfNums(A, B);

        // Printing A+B
        S.o.p("We get:" + sum);
    }
}
```

output:

We get : 90

In the above example, we have 2 methods in package pkg1. One of them is a private method and another one is a public. We are calling the additionOfNums() method from pkg2. This method is in another package and its access specifier is public. So we can access it directly. The additionOfNums() method then calls the sum() method which it has access to because private members can be accessed within a class. In case we called sum() method directly from package pkg2 would have caused an error in the program.

2) Default:

→ A data member, method or class specified with no access modifier is treated as default access modifier by default.

→ Data members, methods or variables having default access modifier can be accessed only within the package.

→ If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

→ It doesn't require any keywords.

→ The scope of the default access specifier is everywhere within the same package and not outside the package.

→ Program: we have created two packages pack and mypack. we are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
// save by A.java
package pack;
class A
{
    void msg()
    {
        S.o.p("Hello");
    }
}

// save by B.java
package mypack;
Import pack.*;

class B
{
    PSVM(String args[])
    {
        A obj = new A(); // Compile Time Error
        obj.msg(); // Compile Time Error
    }
}
```

In the above example, the scope of class A and its method msg is default so it cannot be accessed from outside the package.

→ Program:

```
package abc;
class Demo
{
    void printName()
    {
        S.o.p("Hai");
    }
}

package def;
import abc.*;
class DefaultExample
{
    PSVM(String args[])
    {
        // Since class Demo is not public we cannot access from
           outside the package
        Demo obj = new Demo();
        obj.printName();
    }
}
```

Output:

Compile-time error

3) Protected :

→ Data members or methods or variables declared as "protected" is visible anywhere in its own package but also in subclasses even if they are in another package.

→ A class cannot be protected. protected access modifier in java can be applied on constructor, data members and methods.

→ The protected access modifier is accessible within package and outside the package but through inheritance only.

→ It provides more accessibility than the default modifer.

→ The protected access modifier is specified using the keyword protected.

→ Program: We have created two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A
{
    protected void msg()
    {
        S.o.p("Hello");
    }
}
```

```
//save by B.java
package mypack;
import pack.*;
class B extends A
{
    PSVM(String args[])
    {
        B obj = new B();
        obj.msg();
    }
}
```

output: Hello

→ Program:

Consider a package pkg1 which contains the class below.

```
package pkg1;
public class Calculate
{
    protected int sum(int a, int b)
    {
        return a+b;
    }
}
```

Consider a package pkg2 which contains the class below. It imports the class Calculate from package pkg1.

```
package pkg2;
import pkg1.Calculate;
public class Program extends Calculate
{
    PSVM(String args[])
    {
        int x = 64;
        int y = 13;

        // The add() method is not accessible directly be instantiating the class
        // Calculate cal = new Calculate();
        // int add = cal.sum(x, y);

        // The Calculate class is inherited,
        Program prog = new Program();
        int add = prog.sum(x, y);

        // Printing A+B
        S.O.P("We get: " + add)
    }
}
```

output:

We get: 77

4) Public:

→ Class or method or variable declared as "public" is visible to all classes, whether they are in same package or imported package containing public class.

→ A public class fields and public method can be accessed from any class anywhere, even in different package.

→ The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

→ The public access modifier has the widest scope among all other access modifiers.

→ There is no restriction on the scope of public data members.

→ Program:

```
// save by A.java
package pack;
public class A
{
    public void msg()
    {
        S.o.p("Hello");
    }
}
```

```
// Save by B.java
package mypack;
import pack.*;
class B
{
    PSVM(String args[])
    {
        A obj = new A()
        obj.msg();
    }
}
```

Output: Hello

Program: Consider a package pkg1 which contains the class below.

```
package pkg1;
public class Calculate
{
    public int add(int a, int b);
    {
        return a+b;
    }
}
```

Consider a package pkg2 which contains the class below. It imports the class Calculate from package pkg1.

```
package pkg2;
import pkg1.Calculate;
public class Program
{
    PSVM(String args[])
    {
        int A = 10;
        int B = 15;
        Calculate cal = new Calculate();
        int sum = cal.add(A,B);
        // Printing A+B
        S.O.P("We get: " + sum);
    }
}
```

Output:

We get: 25

In the above example, the package pkg1 contains the class Calculate which has the public access specifier. Due to this specifier, it is being accessed and used in in other packages (pkg2).

Access Specifiers in java

→ Private protected

	Access location	Private		Default	Protected	Public
Same Package	class	Yes	Yes	Yes	Yes	Yes
	sub class	No	Yes	Yes	Yes	Yes
	Non sub class	No	No	Yes	Yes	Yes
Different Package	sub class	No	Yes	No	Yes	Yes
	Non sub class	No	No	No	No	Yes

```
package P1;
class A
{
    private int i;
    int j;
    protected int k;
    public int l;
}
```

Different class but same package	Different package but sub class	Unrelated class but same module	Different module and P1 not exported

Package P1;
class B
{
}
I
A
A
A
	Package P2;
	class C extends A
	{
	}
	I
	I
	A
	A
		package P2;
		class D
		{
		}
		I
		I
		I
		A
			Package x;
			class E
			{
			}
			I
			I
			I
			I

I - Inaccessible

A - Accessible

Packages

packages

- Java API packages (or) Pre-defined (or) Built-in package
  - java.lang
  - java.awt
  - java.Applet
  - java.util
  - java.io
  - java.net

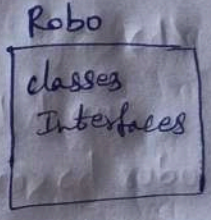
  → import
- User-defined packages

User-defind packages (Udp):

Creating Udp:

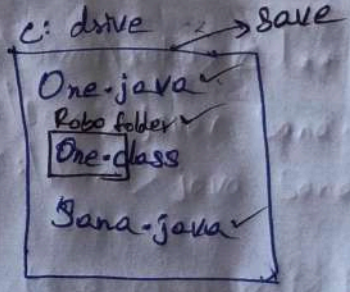
Syntax: package package_name; (package → Reserved keyword)

Ex: package Robo;



→ Program:

```
package Robo;
public class One
{
    public void chitti()
    {
        S.o.p("Hai");
    }
}
```



Compilation:

C:> java -d . One.java

(. → save; One.java → creating in current directory (ie, C: drive); -d → Package declaration creating Robo folder)

```
import Robo.*;   (* → All classes available in this package ie, Robo)
class Sana
{
    psvm(String args[])
    {
        One obj = new One();
        obj.chitti();
    }
}
```

Compilation:

javac Sana.java

Run: java Sana

importing package:

Syntax: import <package_name>;
(reserved keyword)

Ex: import Robo.*;

Program:

```
// Save by Add.java in robo folder
package robo;
public class Add
{
    int x, y;
    public Add()
    {
        x = 100;
        y = 200;
    }
    public void sum()
    {
        S.o.p("Addition of two no's = " + (x+y));
    }
}
```

Folder creation

C:/ drive
↓
Demo folder → Save → Sana.java, Sana.class
↓
robo folder → Save → Add.java, Add.class

```
// Save by Sana.java in Demo folder
import robo.Add;
class Sana
{
    PSVM(String args[])
    {
        Add obj = new Add();
        obj.sum();
    }
}
```

Method-I

```
C:\Users\bogam> cd\          → comes back to C:\
C:\> cd demo
C:\Demo> cd robo
C:\Demo\robo> javac Add.java
C:\Demo\robo> cd..           → comes one folder back
C:\Demo> javac Sana.java
C:\Demo> java Sana           → execution
Addition of two no's = 300
C:\Demo>
```

Method II

```
C:\Users\bogam> cd\
C:\> cd example
C:\> example> javac -d . Add.java
C:\> example> javac Sana.java
C:\> example> java Sana
Addition of two no's = 300
C:\> example>
```

Folder creation

C:/drive
↓
example folder → Sana.class
→ Save → Add.java
→ Sana.java
↓
robo (folder created Automatically) → Add.class

Package:

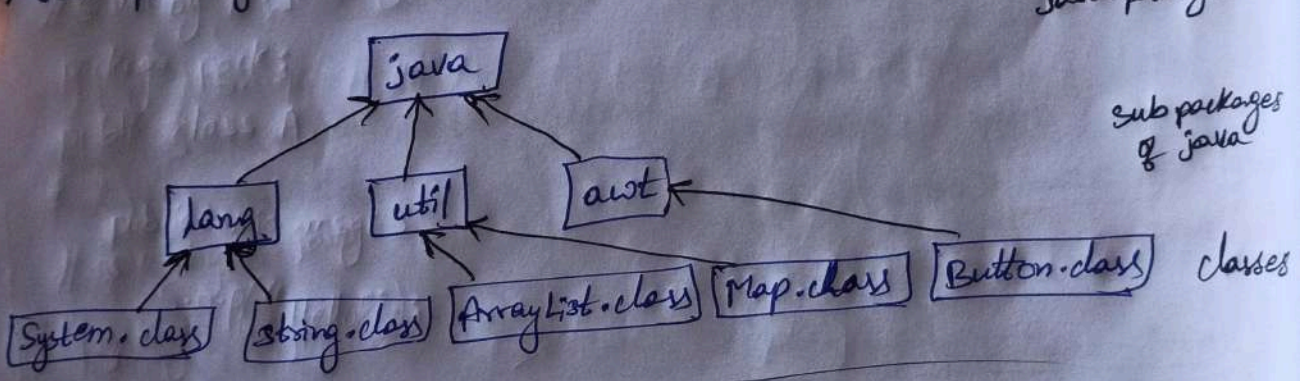
A java package is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two forms, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantages of Java package:

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



Simple example of java package:

The package keyword is used to create a program in java.

```
// save as Simple.java
package mypack;
public class Simple
{
  psvm(String args[])
  {
    S.o.p("Welcome to package");
  }
}
```

How to compile java package:

If you are not using any IDE, you need to follow the syntax given below:

Syntax:

```
javac -d directory javafilename
```

Ex: javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java program program

You need to use fully qualified name ex: mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

output: Welcome to package

The -d is a switch that tells the compiler where to put the class file ie., it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1) import package.*;
2) import package.classname;
3) fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A
{
    public void msg()
    {
        S.o.p("Hello");
    }
}
```

```
//save by B.java
package mypack;
import pack.*;
class B
{
    PSVM(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}
```

Output: Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java
package pack;
public class A
{
    public void msg()
    {
        System.out.println("Hello");
    }
}
```

```
//save by B.java
package mypack;
import pack.A;
class B
{
    public static void main(String args[])
    {
        A obj = new A();
        obj.msg();
    }
}
```

Output: Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name ex: java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A
{
    public void msg()
    {
        S.o.p("Hello");
    }
}

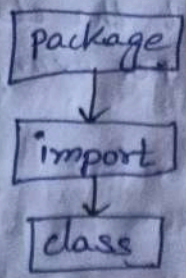
//save by B.java
package mypack;
class B
{ psvm(String args[])
    {
        pack.A obj = new pack.A(); // using fully qualified name
        obj.msg();
    }
}
```

output: Hello

Note: If you import a package, subpackages will not be imported

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackages as well.

Note: sequence of the program must be package then import then class.



Subpackage in java:

Package inside the package is called the subpackage. It should be created to categorize the package further.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group eg. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in net io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package
e.g: com.javatpoint.bean or org.sssit.dao

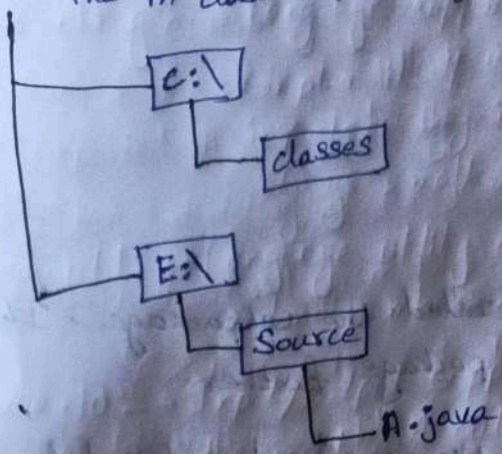
Example of subpackage:

```
package com.javatpoint.core;
class Simple
{
    psvm(String args[])
    {
        S.o.p("Hello subpackage");
    }
}
```

To compile: javac -d . Simple.java
To Run: java com.javatpoint.core.Simple
Output: Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java Source file in classes folder of C: drive. For example:




```
//save as Simple.java
package mypack;
public class Simple
{
  psvm(String args[])
  {
    s.o.p("Welcome to package");
  }
}
```

To compile:

```
e:\source> javac -d c:\classes Simple.java
```

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

```
e:\source> set classpath=c:\classes;.;
e:\source> java mypack.Simple
```

Another way to run this program by classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

output: Welcome to package

Ways to load the class files or jar files:

There are two ways to load the class files temporary and permanent.

* Temporary
  - → By setting the classpath in the command prompt
  - → By -classpath switch
* Permanent
  - → By setting the classpath in the environment variables.
  - → By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
//save as C.java otherwise Compilte Time Error
class A {}
class B{ }
public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
//save as A.java
package javapoint;
public class A{}

//save as B.java
package javapoint;
public class B{}
```

Java System packages and Their classes

Package name	Content
java.lang	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, thread and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, date, etc.
java.io	Input/Output support classes. They provide facilities for the input and output of data.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	classes for creating and implementing applets.

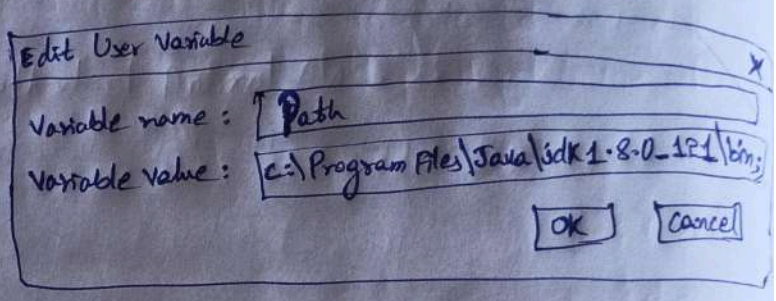
CLASSPATH

CLASSPATH: CLASSPATH is a environmental variable. It stores the location of the folders where java compiler searches for classes that are imported through packages. Usually these classes are contained in the "bin" directory of the java folder. For example, it can be like this

CLASSPATH = C:\Program Files\Java\jdk1.8.0_121\bin;

The first entry is the '.' which indicates current directory and then the other folders follows. When we create our own package, the directory where we store the package must also be included in the CLASSPATH variable.

```
import p.*;
class test
{
 ....
}
```



Running java Program

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

(C:\Program Files\Java\jdk1.8.0_121\bin)

For path variable

Right click on "my computer"
Click on "properties"
click on "Advanced System Setting"
click on "environment variables"
click on "path" for Edit, if there is no path variable then click on "new"

What is path & classpath?

→ Both path and classpath are environment variables in Java.

→ To find the location of files.

path → helps the OS to find the executable files.

classpath → helps the ClassLoader to locate the classfiles.

Note: In IDE's like Netbeans & Eclipse, no need to set path & classpath to compile and run java programs.

Where to set path?

→ JDK is installed in one location (eg: E:\Java), whereas Java-programs are stored in another location (eg: D:\Javacode).

Command Prompt

```
D:\Javacode> javac Test.java
'javac' is not recognized as an internal or external command,
operable program or batch file.

D:\Javacode>
```

Error: OS can't identify the location of javac

How to set path?

In two ways:

1) In command prompt (temporary)
2) Control panel (permanent)

1) Setting path in command prompt: (temporary)

```
D:\Javacode> set path =%path%;E:\Java\bin;
D:\Javacode> javac Test.java
```

2) Setting path - Control Panel: (permanent)

Goto settings → Control Panel\System and security\System\
Advanced System Setting
↓
Environment variables
↓
System variables → path → Edit
↓
variable value: %path%;E:/Java/bin;
↓
OK

(Reopen cmd prompt → run java program)

When to set - Class path?

Java programs & Class files are stored in one location (eg: D:\Javacode), then no need to set classpath.

→ Program: Animal.java

```
package forestpack1;
public class Animal
{
    public void food()
    {
        s.o.p("food method of Animal Class");
    }
    public void sound()
    {
        s.o.p("sound method of Animal Class");
    }
}
```

→ Program: Test.java

```
import forestpack1.Animal;
public class Test
{
    PSVM(String args[])
    {
        Animal tiger = new Animal();
        tiger.food();
        tiger.sound();
    }
}
```

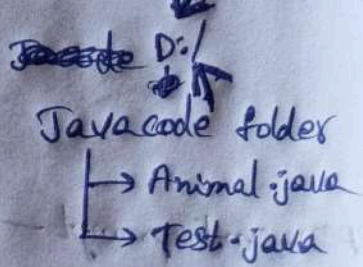
output:

D:\Javacode> javac -d . Animal.java (. → current working directory)

D:\Javacode> javac Test.java

D:\Javacode> java Test

food method of Animal class
sound method of Animal class



Java programs are stored in one location (eg: D:\Javacode), whereas class files are stored in another location (eg: E:\Javaclasses). Then, we must set the class-path variable while executing package programs

Command prompt:

```
D:\Javacode> javac -d E:\Javaclasses Animal.java
D:\Javacode> javac Test.java          (Here classpath is not setted)
D:\Javacode> java Test
Exception in thread "main" java.lang.NoClassDefFoundError: forestpack1\Animal
D:\Javacode>
```

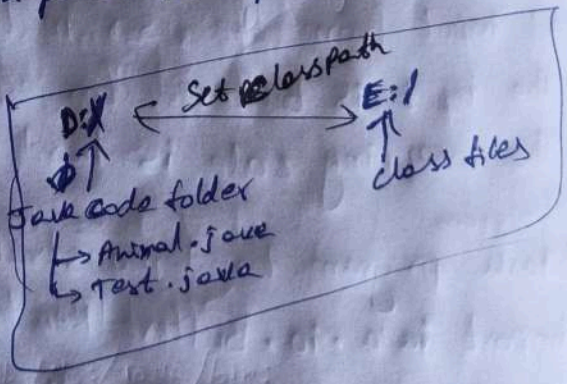
Command prompt:

```
D:\Javacode> javac -d E:\Javaclasses Animal.java
D:\Javacode> set classpath=%classpath%;E:\Javaclasses;
D:\Javacode> javac Test.java
D:\Javacode> java Test
  food method of Animal class
  sound method of Animal class
D:\Javacode>
```

By using classpath, we can store Java programs & package classes in two different locations.

Note:

Classpath can also be set through control panel like path.



case "9". 5

Serialization and Deserialization:

Serialization in simple terms means converting an object into a sequence of bytes, deserialization is exactly the opposite. In deserialization, an object is reconstructed back from the sequence of bytes. In Java, Serialization and deserialization play a great role in the transfer of data and saving it to a database or disk.

1) Overview:

Serialization and deserialization are two import concepts in programming that allows objects to be easily stored, transmitted, and reconstructed. They are used in various scenarios, such as storing objects in a database, sending objects over a network, or caching objects in memory.

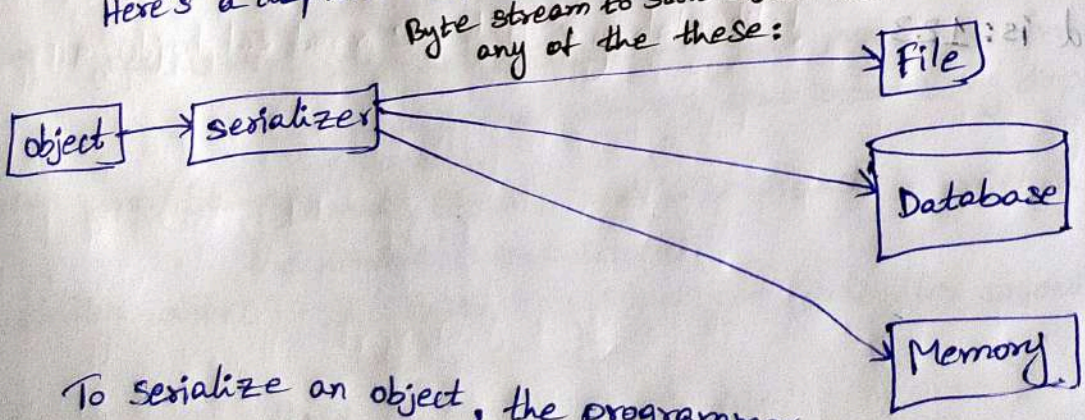
2) Serialization:

An object has three primary characteristics: identity, state and behavior. The state represents the value of data of the object.

Serialization is the process of converting an object's state to a byte stream. This byte stream can then be saved to a file, sent over a network, or stored in a database. The byte stream represents the object's state, which can later be reconstructed to create a new copy of the object.

Serialization allows us to save the data associated with an object and recreate the object in a new location.

Here's a depiction of the serialization process:



To serialize an object, the programmer must first decide on a format and then use the appropriate tools to convert the object to that format.

Serialization Formats:

Many different formats can be used for serialization, such as JSON, XML, and binary. JSON and XML are popular formats for serialization because they are human-readable and can be easily parsed by other systems. Binary formats are often used for performance reasons, as they're typically faster to read and write than text-based formats.

Serialization in Java:

In a java programming language, the Serialization is achieved with the help of interface Serializable. The class whose object needs to be serialized must implement the Serializable interface.

We use the ObjectOutputStream class to write a serialized object to write to a destination. The ObjectOutputStream class provides a method writeObject() to serializing an object.

We use the following steps to serialize an object.

Step 1: Define the class whose object needs to be serialized; it must implement Serializable interface.

Step 2: Create a file reference with file path using FileOutputStream class

Step 3: Create reference to ObjectOutputStream object with file reference.

Step 4: Use writeObject(object) method by passing the object that wants to be serialized.

Step 5: Close the FileOutputStream and ObjectOutputStream.

→ Program:

```
import java.io.*;
public class SerializationExample
{
    psvm(String[] args)
    {
        Student stud = new Student();
        stud.studName = "Rama";
        stud.studBranch = "IT";

        try
        {
            FileOutputStream fos = new FileOutputStream("my_data.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);

            oos.writeObject(stud);
            oos.close();
            fos.close();

            S.o.p("The object has been saved to my_data file!");
        }
        catch(Exception e)
        {
            S.o.p(e);
        }
    }
}
```

Output:

The object has been saved to my_data file!

3) Deserialization:

Deserialization is the reserve process of serialization. It involves taking a byte stream and converting it back into an object. This is done using the appropriate tools to parse the byte stream and create a new object. In Java, the ObjectInputStream class can be used to deserialize a binary format, and the Jackson library can be used to parse a JSON format.

In a java programming language, the Deserialization is achieved with the help of class ObjectInputStream. This class provides a method readObject() to deserializing an object.

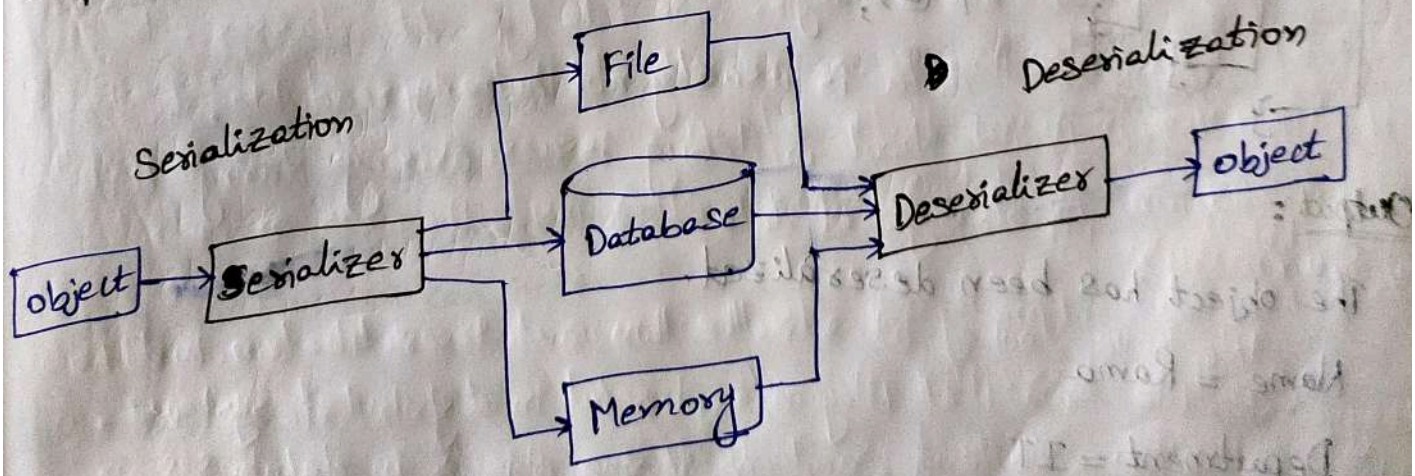
We use the following steps to deserializing an object.

Step 1: Create a file reference with file path in which serialized object is available using FileInputStream class.

Step 2: Create reference to ObjectInputStream object with file reference.

Step 3: Use readObject() method to access serialized object, and typecaste it to destination type.

Step 4: Close the FileInputStream and ObjectInputStream.



→ Program:

```
import java.io.*;
public class DeserializationExample
{
    PSVM(String[] args) throws Exception
    {
        try
        {
            FileInputStream fis = new FileInputStream("my_data.txt");
            ObjectInputStream ois = new ObjectInputStream(fis);

            Student studz = (Student) ois.readObject();
            S.o.p("The object has been deserialized");

            fis.close();
            ois.close();

            S.o.p("Name=" + studz.studName);
            S.o.p("Department=" + studz.studBranch);
        }
        catch(Exception e)
        {
            S.o.p(e);
        }
    }
}
```

Output:

The object has been deserialized.

Name = Rama

Department = IT

4) Storage and Transmission:

Serialization and Deserialization are important in programming because they allows objects to be easily stored and transmitted between different systems. This is especially useful in distributed systems where objects must be transmitted between different machines or in web applications where objects must be sent between a web server and a web browser.

For example, a web application allows user to create and save documents. When a user saves a document, the application needs to store the document's state in a database.

To do this, the application must first serialize the document object to a byte stream and then store the byte stream in the database. Later, when the user wants to open the document, the application must retrieve the byte stream from the database. It first deserializes the byte stream back into a document object and then displays the document to the user.

5) Considerations / Drawbacks:

a) Performance:

Serialization and deserialization can be computationally expensive, especially for large or complex objects. Converting objects to bytes and back can take a significant amount of time and resources, which can impact the system's performance.

b) Platform and Language Dependencies:

Serialization and deserialization can be platform and language dependent. Different programming languages and platforms may have their own implementations of serialization and deserialization, which can lead to compatibility issues when transmitting data between different systems.

c) Versioning:

If the format of the serialized data changes over time, it can lead to versioning issues. For example, suppose a new version of an application is released with a different data format. In that case, older versions may not be able to deserialize the data correctly, leading to errors or data loss.

d) Unserializable Objects:

It's also worth nothing that not all objects can be serialized. Some objects may contain resources that cannot be serialized, like file handles or network sockets.

e) Security:

It's important to be aware of the security risks associated with serialization and deserialization, as maliciously crafted byte streams can be used to exploit vulnerabilities in a program.

To mig mitigate this risk, it's best practise to use a well-vetted library for serialization and deserialization and only to deserialize data from trusted sources.

f) Limited human-readability:

Serialized data is usually not human-readable, making it difficult to troubleshoot or debug issues arising during the transmission or storage of data.

6) Summary:

We have looked at serialization and deserialization and how they work. We saw that they are useful in distributed systems, web applications, caching, and in storing objects in databases or other data stores.

However, it's important to know the security risks of ~~serial~~ serialization and deserialization. Remember also to use well-vetted libraries and only deserialize data from trusted sources.

→ Program: (Serialization and deserialization)

```
import java.io.*;
public class SerializationTest implements Serializable
{  int a=1, b=2;
   transient int c=3;
   psvm(String[] args) throws Exception
   {
      SerializationTest obj = new SerializationTest();
      //serialization
      FileOutputStream fos = new FileOutputStream("hello.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(obj);
      //de-serialization
      FileInputStream fis = new FileInputStream("hello.txt");
      ObjectInputStream ois = new ObjectInputStream(fis);
      SerializationTest test = (SerializationTest) ois.readObject();
      S.o.p("a=" + test.a);
      S.o.p("b=" + test.b);
      S.o.p("c=" + test.c);
   }
}
```

Output:

a=1
b=2
c=0

transient Keyword:

While serializing an object, if we don't want certain data members of the object to be serialized we can mention it transient. transient keyword will prevent the data member from being serialized.

Enumeration:

An enumeration (enum for short) in Java is a special data type which contains a set of predefined constants.

You will usually use an enum when dealing with values that aren't required to change, like days of the weeks, ~~seasons~~ seasons of the year, colors, and so on.

→ The Enum in Java is a data type which contains a fixed set of constants.

→ It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY), directions (NORTH, SOUTH, EAST, and WEST), seasons (SPRING, SUMMER, WINTER, and AUTUMN or FALL), ~~colors~~ colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

→ Java Enums can thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and finaly implicity. It is available since JDK 1.5.

→ Enums are used to create our own data type like classes. The enum data type (also Known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class or outside the class.

→ Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java Enum:

→ Enum improves type safety.
→ Enum can be easily used in switch.
→ Enum can be traversed.
→ Enum can have fields, constructors and methods.
→ Enum may implement many interfaces but cannot extend any class because it internally extends Enum class.

→ **Program:**

```
class EnumExample
{
  // defining the enum inside the class
  public enum Season { WINTER, SPRING, SUMMER, FALL }
  // main method
  PSVM(String[] args)
  {
    // traversing the enum
    for(Season s : Season.values())
    {
      S.o.p(s);
    }
  }
}
```

output:

```
WINTER
SPRING
SUMMER
FALL
```

Java enum where we are using value(), valueOf(), and ordinal() methods of Java enum.

→ Program:

```
class EnumExample
{
  //defining enum within class
  public enum Season{WINTER, SPRING, SUMMER, FALL}
  //creating the main method
  psvm (String[] args)
  {
    //printing all enum
    for(Season s : Season.values())
    {
      s.o.p(s);
    }
    s.o.p("Value of WINTER is:" + Season.valueOf("WINTER"));
    s.o.p("Index of WINTER is:" + Season.valueOf("WINTER").
                                  ordinal());
    s.o.p("Index of SUMMER is:" + Season.valueOf("SUMMER").
                                  ordinal());
  }
}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```

Note: Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.

What is the purpose of the values() method in the enum?

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

What is the purpose of the valueOf() method in the enum?

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

What is the purpose of the ordinal() method in the enum?

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

Define Java Enum:

The enum can be defined within or outside the class because it is similar to a class. The semicolon (;) at the end of the enum constants are optional. For example:

```
enum Season {WINTER, SPRING, SUMMER, FALL}

(or)

enum Season{WINTER, SPRING, SUMMER, FALL;}
```

Both the definitions of Java enum are the same.

→ Program: Enum defined outside class:

```
enum Season{WINTER, SPRING, SUMMER, FALL}
class EnumExample
{
    PSVM(String[] args)
    {
        Season s = Season.WINTER;
        S.o.p(s);
    }
}
```

Output:

```
WINTER
```

Example:

→ Program: Enum defined inside class:

```
class EnumExample
{
    enum Season { WINTER, SPRING, SUMMER, FALL; } //semicolon(;) is optional
    PSVM(String[] args)
    {
        Season s = Season.WINTER; //enum type is required to access WINTER
        S.o.p(s);
    }
}
```

Output:

WINTER

Initializing specific values to the enum constants:

The enum constants have an initial value which starts from 0, 1, 2, 3, and so on. But, we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors, and methods.

→ Program: Example of specifying initial value to the enum constants:

```
class EnumExample
{
    enum Season
    {
        WINTER(5), SPRING(10), SUMMER(15), FALL(20);
        private int value;
        private Season(int value)
        {
            this.value = value;
        }
    }
    PSVM(String args[])
    {
        for(Season s : Season.values())
        S.o.p(s + " " + s.value);
    }
}
```

Output:

```
WINTER 5
SPRING 10
SUMMER 15
FALL 20
```

→ Note: Constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

Java Enum in a switch statement:

→ Program:

```
class EnumExample
{
  enum Day{SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
  PSVM(String args[])
  {
    Day day = Day.MONDAY;

    switch (day)
    {
      case SUNDAY:
      S.o.p("sunday");
      break;

      case MONDAY:
      S.o.p("monday");
      break;

      default:
      S.o.p("other day");
    }
  }
}
```

Output:

monday

Autoboxing and Unboxing:

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object Wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called unboxing.

In Java, primitive data types are treated differently so do there comes the introduction of wrapper classes where two components play a role namely Autoboxing and Unboxing. Autoboxing refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

→ Passed as a parameter to a method that excepts an objects of the corresponding wrapper class.
→ Assigned to a variable of the corresponding wrapper class.

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

→ Passed as a parameter to a method that excepts a value of the corresponding primitive type.
→ Assigned to a variable of the corresponding primitive type.

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

autoboxing: Primitive datatype to object wrapper class.

Unboxing: object wrapper class to Primitive datatype

The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing. Now let us discuss a few advantages of autoboxing and unboxing in order to get why we are using it.

→ Autoboxing and unboxing lets developers write cleaner code, making it easier to read.

→ The technique lets us use primitive types and Wrappers class objects interchangeably and we do not need to perform any typecasting explicity.

→ Program: Autoboxing (converting primitive Datatypes into object)

```
import java.lang.*;
class wrapperclass
{
    PSVM(String args[])
    {
        int a = 100;
        Integer obj1 = new Integer(a);
        //Boxing process with constructors
        Integer obj2 = Integer.valueOf(a);
        // Boxing using valueOf() method
        Integer obj3 = a; //autoboxing
        S.o.p(obj1 + " " + obj2 + " " + obj3);
    }
}
```

100 stores

parameter

obj1 → 100

obj2 → 100

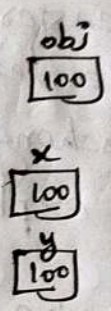
obj3 → 100

output:

100 100 100

→ **Program:** Unboxing (convert object type to primitive datatype).

```
import java.lang.*;
class wrapperclass
{
    PSVM(String args[])
    {
        int a = 100;
        Integer obj = new Integer(a);
        //Using intValue() method
        int x = obj.intValue();
        int y = obj;
        S.o.p(a+" "+x+" "+y+" "+obj);
    }
}
```



output:

100 100 100 100

Generics

→ Generics means parameterized types. The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics?

→ The Object is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature.

→ Generics in Java are similar to templates in C++. For example, classes like HashSet, ArrayList, HashMap, etc., use generics very well. There are some fundamental differences between the two approaches to generic types.

Types of Java Generics:

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

Generic classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, seperated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Generic class:

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type> ()
```

Note: In Parameter type we can not use primitives like 'int', 'char' or 'double'.

→ A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Let's see a simple example to create and use the generic class.

Creating a generic class:

```
class MyGen<T>
{
    T obj;
    void add(T obj)
    {
        this.obj = obj;
    }
    T get()
    {
        return obj;
    }
}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class:

Let's see the code to use the generic class.

```
class TestGenerics
{
    psvm(String args[])
    {
        MyGen<Integer> m = new MyGen<Integer>();
        m.add(2);
        // m.add("Konica"); // Compile time error
        S.o.p(m.get());
    }
}
```

output: 2

→ Program:

```
// We use < > to specify Parameter type.
class Test <T>
{
    // An object of type T is declared
    T obj;
    // Constructor --> we can use int, float, double, char, String etc
    Test (T obj)
    {
        this.obj = obj;
    }
    public T getObject()
    {
        return this.obj;
    }
}

// Driver class to test above
class Main
{
    PSVM(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        S.o.p(iObj.getObject());

        // instance of String type
        Test<String> sObj = new Test<String>("Konica");
        S.o.p(sObj.getObject());
    }
}
```

Output:

15

Konica

→ Removing code Duplication

renaming integer & String method and write it as only one method by using Generics <T> class.

```
class Test
{
    public int IntegerValue(int num1)
    {
        return num1;
    }
    public String StringValue(String num2)
    {
        return num2;
    }
}
class Main
{
    psvm(String args[])
    {
        Test tt = new Test();
        S.o.p(tt.IntegerValue(15));
        S.o.p(tt.StringValue("Konica"));
    }
}
```

→ Program: (use c

// We can use

```
class Test <
{
    T obj1;
    U obj2;
    // const
    Test(
    {
    }
    // To
    pub
    {
    }
}
// Drive
class
{
}
```

Output:

Konic

15

→ Program: (We can also pass multiple Type parameter in Generic classes.)

```
// We can use <> to specify Parameter type
class Test <T, U>
{
    T obj1; // An object of type T
    U obj2; // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        S.o.p(obj1);
        S.o.p(obj2);
    }
}

// Driver class to test above
class Main
{
    PSVM(String[] args)
    {
        Test <String, Integer> obj = new Test <String, Integer> ("Konica", 15);

        obj.print();
    }
}
```

output:

```
Konica
15
```

Type Parameters in Java Generics:

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

T - Type
E - Element
K - Key
N - Number
V - Value

Advantages of Generics:

Programs that use Generics has got many benefits over non-generic code.

1) Code Reuse: we can write a method/class/interface once and use it for any type we want.

2) Type Safety: Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Generic Method:

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Let's see a simple example of java generic method to print array elements. We are using here E to denote the element.

→ Program:

```
public class TestGenerics
{
    public static <E> void printArray (E[] elements)
    {
        for (E element : elements)
        {
            S.o.p(element);
        }
        S.o.p();
    }

    psvm (String args[])
    {
        Integer[] intArray = {10, 20, 30, 40, 50};
        Character[] charArray = {'K', 'O', 'N', 'I', 'C', 'A'};

        S.o.p("Printing Integer Array");
        printArray(intArray);

        S.o.p("Printing Character Array");
        printArray(charArray);
    }
}
```

Output:

```
Printing Integer Array
10
20
30
40
50
Printing Character Array
K
O
N
I
C
A
```

Wildcard in Java Generics:

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Numbers class through any child class object.

We can use a wildcard as a type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

→ Program:

```
import java.util.*;
abstract class Shape
{
    abstract void draw();
}
class Rectangle extends Shape
{
    void draw()
    {
        S.o.p("drawing rectangle");
    }
}
class Circle extends Shape
{
    void draw()
    {
        S.o.p("drawing circle");
    }
}
class GenericTest
{
    //creating a method that accepts only child class of shape
    public static void drawShapes(List<? extends Shape> lists)
    {
        for(Shape s:lists)
        {
            s.draw(); //calling method of Shape class by child class instance
        }
    }
```

```
psvm (String args[])
{
    List<Rectangle> list1 = new ArrayList<Rectangle>();
    list1.add(new Rectangle());

    List<Circle> list2 = new ArrayList<Circle>();
    list2.add(new Circle());
    list2.add(new Circle());

    drawShapes(list1);
    drawShapes(list2);
}
```

output:

drawing rectangle
drawing circle
drawing circle

Unit -3

Exception Handling in Java:

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

What is Exception in Java?

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling:

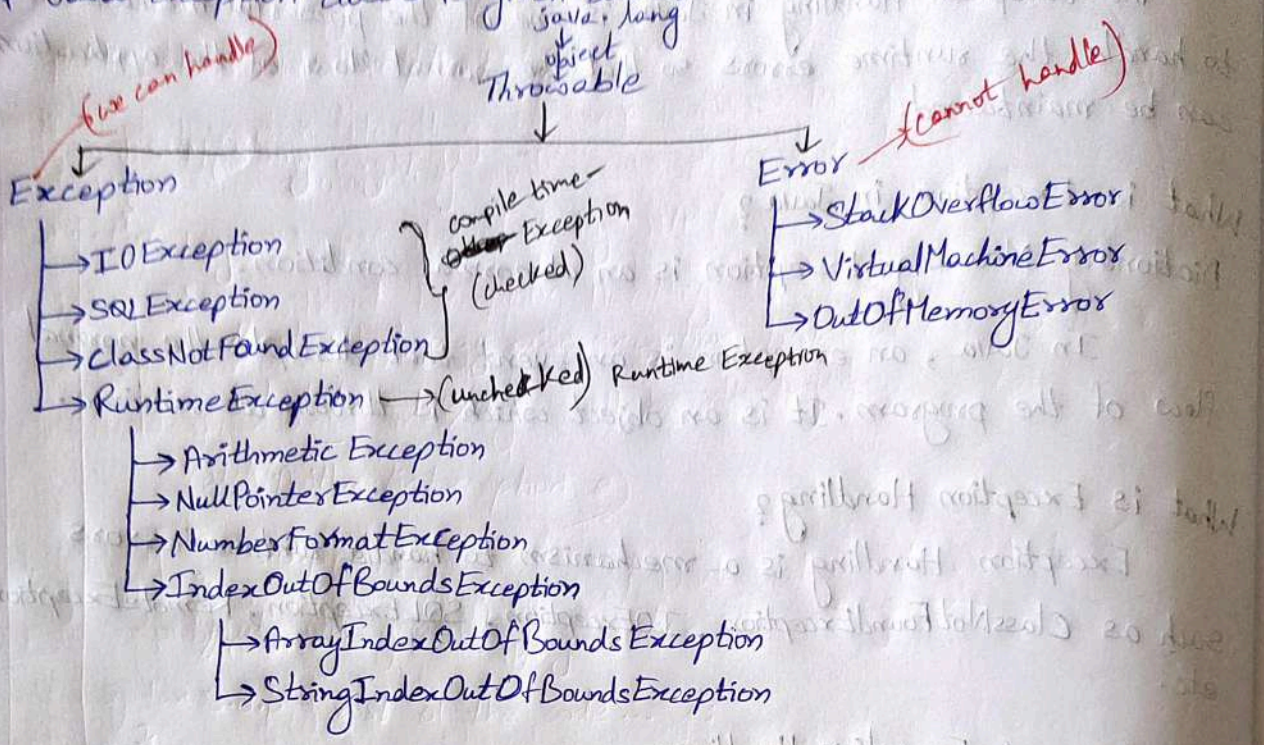
The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5; //exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, ie., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes:

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions:

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1) Checked Exception
2) Unchecked Exception
3) Error

Difference between Checked and Unchecked Exceptions:

1) Checked Exception:

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, FileNotFoundException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception:

The classes that inherit (extends) the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error:

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords:

Java provides five keywords that are used to handle the exception. The following table describes each.

try - The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

catch - The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

finally - The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

throw - The "throw" keyword is used to throw an exception.

throws - The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example:

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

```
public class JavaExceptionExample
{
    PSVM(String args[])
    {
        try
        {
            //code that may raise exception
            int data = 100/0;
        }
        catch(ArithmeticException e)
        {
            S.o.p(e);
        }
        //rest code of the program
        S.o.p("rest of the code...");
    }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException: / by zero
rest of the code...
```

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions:

There are given some scenarios where unchecked exception may occur. They are as follows:

1) A scenario where ArithmeticException occurs:

If we divide any number by zero, there occurs an ArithmeticException.

~~int a=50/0~~

```
int a = 50/0; //ArithmeticException
```

$\frac{Something(n)}{zero(0)} = infinity(\infty)$

2) A scenario where NullPointerException occurs:

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s = null;
S.o.p(s.length());  //NullPointerException
```

3) A scenario where NumberFormatException occurs:

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
String s = "abc";
int i = Integer.parseInt(s);  //NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundException occurs:

When an array exceeds to it's size, the ~~ArrayIndexOutOf~~ ArrayIndexOutOfBoundsException occurs. ~~There~~ There may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
int a[] = new int[5];
a[10] = 50;  //ArrayIndexOutOfBoundsException
```

```
int x[] = {10, 20, 30};
S.o.P(x[5]);  //ArrayIndexOutOfBoundsException
```

Exception Handling:

→ Exception is a runtime error. (or)
→ Any abnormal event in a program is called an Exception.
→ Exception Handling is a mechanism it process the generated exception. (It handle runtime error).
→ Through Exception Handling we can ensure the program is not terminated abnormally and users data not lost.
→ Runtime errors are detected by JVM (Java Virtual Machine).

Note: In Java without Exception Handling '0' will generated the exception and then program will be terminated. No user understandable message will be displayed.

1) Try – It is a keyword used to create block statements, which are doubtfull of generating exception.

Syntax:
```
try
{
    ==== doubtfull
}
```

2) Catch – It is a keyword which is used to handle the exception. It contains block statements (handling code).

Syntax:
```
catch(TypeofException exceptionobject)
{
    ==== handling code
}
```

→ Example:

```
try
{
    ==== doubtfull code
}
catch(TypeofException ExceptionObject)
{
    ==== handling code
}
```

$c = b/a = 20/10 = 2$

$c = b/a = 0/10 = 0$

$c = b/a = 10/0 = \infty$ exception

→ Program:

```
import java.lang.*;
class Robo
{
    PSVM(String args[])
    {
        int x=10, y=0, z;
        try
        {
            z = x/y; // Divide by zero
            S.o.p(z);
        }
        catch(ArithmeticException e)
        {
            S.o.p("Division by Zero Exception");
        }
        S.o.p("Program over");
    }
}
```

Output:

```
Division by Zero Exception
Program over
```

Exception types: (Types of Exceptions)

Exception: An Exception is a condition that is caused by a runtime error.

→ There are 3 types of Exceptions.

1) Checked Exception: The Exception which are detected at compile time.
→ The classes that extend Throwable class except RuntimeException and Error are known as checked Exception.
Ex: IOException, SQLException, FileNotFoundException ..... etc.

2) Unchecked Exception: The Exception which are detected at runtime.
→ The classes that extends RuntimeException are known as Unchecked Exception.
Ex: ArithmeticException, NullPointerException .... etc

3) Error: Error is irrecoverable.
Ex: OutofMemory, VirtualMachineError, AssertionError ... etc

Note: 1) Every Exceptions are also represented by classes in Java.
2) All exceptions are subclasses of exception class.

Built-in Exceptions:

→ Which are available as part of java.lang package.

1) ArithmeticException: Runtime error during Arithmetic operations such as Division by zero.

2) ArrayIndexOutOfBoundException: Referring to the position of the element that is not in an array.

3) StringIndexOutOfBoundException: Referring charactor position in the string that is not found.

4) NullPointerException: Referring to the method which Contains empty object (NULL).

5) FileNotFoundException: File not found in the directory.

6) IOException: General failures such as in ability to read from a file.

7) NumberformatException: Passing one format instead of another where a conversion between string and number.

8) InterruptedException: One processor interrupt another processor.

9) NoSuchMethodException: Calling a method which is not available.

10) ArrayStorageException: These are the exceptions created by the user.

Termination or Presumptive models:

Termination Model:

In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.

In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

Resumptive Model:

The alternative of termination model is resumptive model. In resumptive model, the exception handler is excepted to do something to stable the situation, and then the faulting method is retried. In resumptive model we hope to continue the execution after the exception is handled.

In resumptive model we may use a method call that want resumption like behavior. We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.

Different Models:

→ Resumptive: Resume at point of occurence.
→ Termination: Causes block to terminate.
→ Die: Causes entire program to exit.
→ Try: Uses either the Resumptive or Termination ~~block~~ model.

throw:

→ It is a Keyword used to create an exception and throw it explicitly.

→ Throw our own exception using throw Keyword.

→ The flow of execution stops immediatly after the throw statements and any subsequences are not executed.

Syntax:

```
throw new Exceptiontype("content");
```

- `throw` → Keyword
- `new` → operator (creating dynamic object)
- `Exceptiontype` → Any type of exception.
- `"content"` →

→ **Program:**

```
import java.lang.*;
class Robo
{
    PSVM(String args[])
    {
        try
        {
            throw new ArithmeticException("This is my own Exception");
            S.o.p("This is try block");
        }
        catch(ArithmeticException e)
        {
            S.o.p(e.getMessage());
        }
        S.o.p("Program over");
    }
}
```

Output: This is my own Exception

~~This is my own~~

Program over

Java throw Exception:

In Java, exception allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw Keyword:

The Java throw keyword is used to throw an exception explicitly.

We specify the exception object which is to be thrown. The exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance ie,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub-class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw Keyword Example:

Example 1: Throwing Unchecked Exception

In this example, we have created a method name validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate methode that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1
{
    //function to check if person is eligible to vote or not
    public static void validate (int age)
    {
        if (age < 18)
        {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException ("Person is not eligible to vote");
        }
        else
        {
            S.o.p ("Person is eligible to vote!!");
        }
    }

    PSVM (String args[])
    {
        // calling the function
        validate (13);
        S.o.p ("rest of the code....");
    }
}
```

output:

Person is not eligible to Vote
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to Vote
~~rest of the code~~

The above code throws an unchecked exception. Similarly, coe can also throw unchecked and used defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Example 2: Throwing checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
import java.io.*;
public class TestThrow2
{
  //function to check if person is eligible to vote or not
  public static void method() throws FileNotFoundException
  {
    FileReader file = new FileReader("c:\\Users\\Anwarati\\Desktop\\abc.txt");
    BufferedReader fileInput = new BufferedReader(file);

    throw new FileNotFoundException();
  }
  PSVM(String args[])
  {
    try
    {
      method();
    }
    catch(FileNotFoundException e)
    {
      e.printStackTrace();
    }
    S.o.p("rest of the code...");
  }
}
```

Output:

```
java.io.FileNotFoundException
rest of the code...
```

Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

TestThrow3.java

```
//class represents user-defined exception
class UserDefinedException extends Exception
{
    public UserDefinedException(String str)
    {
        //Calling constructor of parent Exception
        super(str);
    }
}

//class that uses above MyException
public class TestThrow3
{
    PSVM (String args[])
    {
        try
        {
            //throw an object of user defined exception
            throw new UserDefinedException("This is user-defined exception");
        }
        catch (UserDefinedException ude)
        {
            S.o.p ("Caught the exception");
            //Print the message from MyException object
            S.o.p (ude.getMessage());
        }
    }
}
```

output:

```
Caught the exception
This is user-defined exception
```

throw:

The throw Keyword is used to throw an exception instance explicitly from a try block to corresponding catch block. That means it is used to transfer the control from try block to corresponding catch block.

The throw Keyword must be used inside the try block. When JVM encounters the throw keyword, it stops the execution of try block and jump to the corresponding catch block.

- Using throw keyword only object of Throwable class or its sub classes can be thrown.
- Using throw keyword only one exception can be thrown.
- The throw Keyword must followed by an throwable instance.

The following is the general syntax for using throw keyword in a try block.

syntax: throw instance;

Here the instance must be throwable instance and it can be created dynamically using new operator.

→ Program:

```
import java.util.Scanner;
public class Sample
{
    psvm(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int num1, num2, result;
        S.o.print("Enter any two numbers:");
        num1 = input.nextInt();
        num2 = input.nextInt();
        try
        {
            if(num2 == 0)
            {
                throw new ArithmeticException("Division by zero is not possible");
            }
            result = num1/num2;
            S.o.p(num1 + "/" + num2 + "=" + result);
        }
        catch(ArithmeticException ae)
        {
            S.o.p("Problem info:" + ae.getMessage());
        }
        S.o.p("End of the program");
    }
}
```

Output:

```
Enter any two numbers: 10 0
Problem info: Division by zero is not possible
End of the program
```

throws:

→ It is a keyword, It contains list the type of exception that a may thrown.

→ throws keyword is used when we doesn't want to handle the exception and try to send the exception to the JVM or caller.

Syntax: returntype methodname(parameters) throws Exceptionslist
{
}

→ Program:

```
import java.lang.*;
class Robo
{
    static void chitti() throws ArithmeticException
    {
        int x=10;
        int y=0;
        int res = x/y;
        S.o.p(res);
    }
    PSVM(String args[])
    {
        try
        {
            chitti();
        }
        catch(ArithmeticException e)
        {
            S.o.p("Division by Zero");
        }
    }
```

Output: Division by Zero

Java throws Keyword:

The Java throws Keyword is used to declare an exception. It gives information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmer's fault that he is not checking the code before it being used.

Syntax:

```
return_type method_name() throws exception_class_name
{
    // method code
}
```

Which exception should be declared?

checked exception only, because:

→ Unchecked exception: under our control so we can correct our code.

→ error: beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantages of Java throws Keyword:

Now checked Exception can be propogated (forwarded in call stack). It provides information to the caller of the method about the exception.

Java throws Example:

Let's see the example of Java throws clause which describes that checked exception can be propagated by throws keyword.

Testthrows1.java

```
import java.io.IOException;
class Testthrows1
{
    void m() throws IOException
    {
        throw new IOException("device error"); //checked exception
    }
    void n() throws IOException
    {
        m();
    }
    void p()
    {
        try
        {
            n();
        }
        catch(Exception e)
        {
            S.o.p("exception handled");
        }
    }
    PSVM(String args[])
    {
        Testthrows1 obj = new Testthrows1();
        obj.p();
        S.o.p("normal flow....");
    }
}
```

Output: exception handled
normal flow....

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

There are two cases:

1) Case 1: We have caught the exception ie., we have handled the exception using try/catch block.
2) Case 2: We have declared the exception ie., specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block:

In case we handle the exception, the code will be executed fine whether occurs during the program or not.

1. stthrows2.java

```
import java.io.*;
class M
{
  void method() throws IOException
  {
    throw new IOException("device error");
  }
}
public class Testthrows2
{
  psvm(String args[])
  {
    try
    {
      M m = new M();
      m.method();
    }
    catch(Exception e)
    {
      S.o.p("exception handled");
    }
    S.o.p("normal flow...");
  }
}
```

Output: exception handled
normal flow...

Case 2: Declare Exception

→ In case we declare the exception, if exception does not occur, the code will be executed fine.

→ In case we declare the exception and the exception occurs, it will be thrown at runtime because throws does not handle the exception.

Let's see examples for both the scenario.

A) If exception does not occur

Testthrows3.java

```
import java.io.*;
class M
{
    void method() throws IOException
    {
        S.o.p("device operation performed");
    }
}
class Testthrows3
{
    PSVM(String args[]) throws IOException // declared exception
    {
        M m = new M();
        m.method();
        S.o.p("normal flow...");
    }
}
```

Output: device operation performed
normal flow...

B) If exception occurs

Testthrows4.java

```
import java.io.*;
class M
{
    void method() throws IOException
    {
        throw new IOException("device error");
    }
}
class Testthrows4
{
    PSVM(String args[]) throws IOException //declare exception
    {
        M m = new M();
        m.method();
        S.o.p("normal flow...");
    }
}
```

Output:

Exception in thread "main" java.io.Exception : device error

Difference between throw and throws:

throw	throws
→ It is used to throw an exception object explicitly.	→ It is used to declare an exception as well as by pass to the caller (JVM).
→ It is always present inside method body.	→ It is always with method signature.

→ void robo
{
throw new ArithmeticException("Hai");
}
	→ void robo() throws ArithmeticException, List,
	{
	}?

→ we can throw only one Exception at a time.	→ we can handle multiple Exceptions using throws.
→ checked Exception cannot be handled using throw.	→ checked and Unchecked Exceptions can be handle using throws.

Uncaught Exceptions:

In Java, assume that, if we do not handle the exceptions in a program. In this case, when an exceptions occurs in a particular function, then Java prints a exception message with the help of uncaught exception handler.

The uncaught exceptions are the exceptions that are not caught by the compiler but automatically caught and handled by the Java built-in exception handler.

Java programming language has a very strong exception handling mechanism. It allows us to handle the exception use the keywords like try, catch, finally, throw and throws.

When an uncaught exception occurs, the JVM calls a special private method known dispatchUncaughtException(), on the Thread class in which the exception occurs and terminates the thread.

The Division by zero exception is one of the example for uncaught exceptions.

→ Program:

```
import java.util.Scanner;
public class UncaughtExceptionExample
{
    psvm(String[] args)
    {
        Scanner read = new Scanner(System.in);
        S.o.p("Enter the a and b values:");
        int a = read.nextInt();
        int b = read.nextInt();
        int c = a/b;
        S.o.p(a + "/" + b + "=" + c);
    }
}
```

Output: Enter the a and b values:
10
0
Exception in thread "main" java.lang.ArithmeticException: / by zero

In the above example code, we are not used try and catch blocks, but when the value of b is zero the division by zero exception occurs and it caught by the default exception handler.

try and catch block:

try block:

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch for finally block.

Syntax of Java try-catch:

```
try
{
  // code that may throw an exception
}
catch (Exception_class_Name ref)
{
}
```

Syntax of try-finally block:

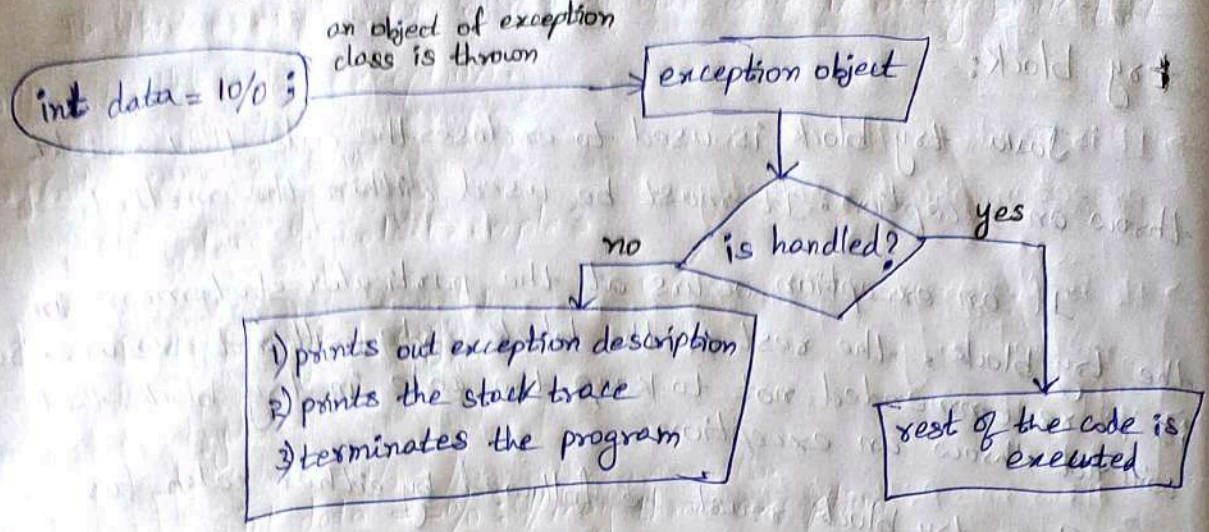
```
try
{
  // code that may throw an exception
}
finally
{
}
```

catch block:

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (ie., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block:



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

→ Prints out exception description.
→ Prints the stack trace (Hierarchy of methods where the exception occurred).
→ Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, ie., rest of the code is executed.

Problem without exception handling:

Let's try to understand the problem if we don't use a try-catch block.

Example 1: {TryCatchExample1.java

```
public class TryCatchExample1
{
    psvm(String[] args)
    {
        int data = 50/0 ; // may throw exception
        S.o.p("rest of the code");
    }
}
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

As displayed in the above example, the rest of the code is not executed (in such case, the rest of the code statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling:

Let's see the solution of the above problem by a java try-catch block.

Example 2: TryCatchExample2.java

```
public class TryCatchExample2
{
    PSVM(String[] args)
    {
        try
        {
            int data = 50/0; //may throw exception
        }
        //handling the exception
        catch(ArithmeticException e)
        {
            S.o.p(e);
        }
        S.o.p("rest of the code");
    }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

As displayed in the above example, the rest of the code is executed, i.e., the "rest of the code" statement is printed.

Example 3:

In this example, we also kept the code in a try block that will not throw an exception.

TryCatchExample3.java

```
public class TryCatchExample3
{
    PSVM(String[] args)
    {
        try
        {
            int data = 50/0; //may throw exception
                //if exception occurs, the remaining statement will not execute.
            S.o.p("rest of the code");
        }
        //handling the exception
        catch(ArithmeticException e)
        {
            S.o.p(e);
        }
    }
}
```

Output:

java.lang.ArithmeticException: / by zero

Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4:

Here, we handle the exception using the parent class exception.

TryCatchExample4.java

```
public class TryCatchExample4
{
    PSVM(String[] args)
    {
        try
        {
            int data = 50/0; // may throw exception
        }
        // handling the exception by using Exception class
        catch(Exception e)
        {
            S.o.p(e);
        }
        S.o.p("rest of the code");
    }
}
```

output: java.lang.ArithmeticException: / by zero
rest of the code

Example 5:

Let's see an example to print a custom message on exception.

TryCatchExample5.java

```
public class TryCatchExample5
{
    PSVM(String[] args)
    {
        try
        {
            int data = 50/0; // may throw exception
        }
        // handling the exception
        catch (Exception e)
        {
            // displaying the custom message
            S.o.p("Can't divided by zero");
        }
    }
}
```

output: Can't divided by zero

Example 6:

Let's see an example to resolve the exception in a catch block.

TryCatchExample6.java

```
public class TryCatchExample6
{
    psvm(String[] args)
    {
        int i=50;
        int j=0;
        int data;
        try
        {
            data = i/j; // may throw exception
        }
        //handling the exception
        catch(Exception e)
        {
            //resolving the exception in catch block
            S.o.p(i/(j+2));
        }
    }
```

Output: 25

Example 7:

In this example, along with try block, we also enclose exception code in a catch block.

TryCatchExample7.java

```
public class TryCatchExample7
{
    psvm(String[] args)
    {
        try
        {
            int data1 = 50/0; // may throw exception
        }
        catch(Exception e)
        {
            //generating the exception in catch block
            int data2 = 50/0; // may throw exception
        }
        S.o.p("rest of the code");
    }
}
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8:

In this example, we handle the generated exception (ArithmeticException) with a different type of exception class (ArrayIndexOutOfBoundException).

TryCatchExample8.java

```
public class TryCatchExample8
{
    PSVM(String[] args)
    {
        try
        {
            int data = 50/0; //may throw exception
        }
        //try to handle the ArithmeticException using ArrayIndexOutOfBoundException
        catch(ArrayIndexOutOfBoundException)
        {
            S.o.p(e);
        }
        S.o.p("rest of the code");
    }
}
```

output: Exception in thread "main" java.lang.ArithmeticException: / by zero

Example 9:

Let's see an example to handle another unchecked exception.

TryCatchExample9.java

```
public class TryCatchExample9
{
    PSVM(String[] args)
    {
        try
        {
            int arr[] = {1,3,5,7};
            S.o.p(arr[10]); //may throw exception
        }
        //handling the array exception
        catch(ArrayIndexOutOfBoundException e)
        {
            S.o.p(e);
        }
        S.o.p("rest of the code");
    }
}
```

Output:

java.lang.ArrayIndexOutOfBoundException: 10
rest of the code.

Example 10:

Let's see an example to handle checked exception.

TryCatchExample10.java

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10
{
    PSVM(String[] args)
    {
        PrintWriter pw;
        try
        {
            pw = new PrintWriter("jtp.txt"); may throw exception
            pw.println("saved");
        }
        // providing the checked exception handler
        catch(FileNotFoundException e)
        {
            S.O.P(e);
        }
        S.O.P("File saved successfully");
    }
}
```

Output: File saved successfully

Multiple catch clauses: (Java catch Multiple Exceptions)

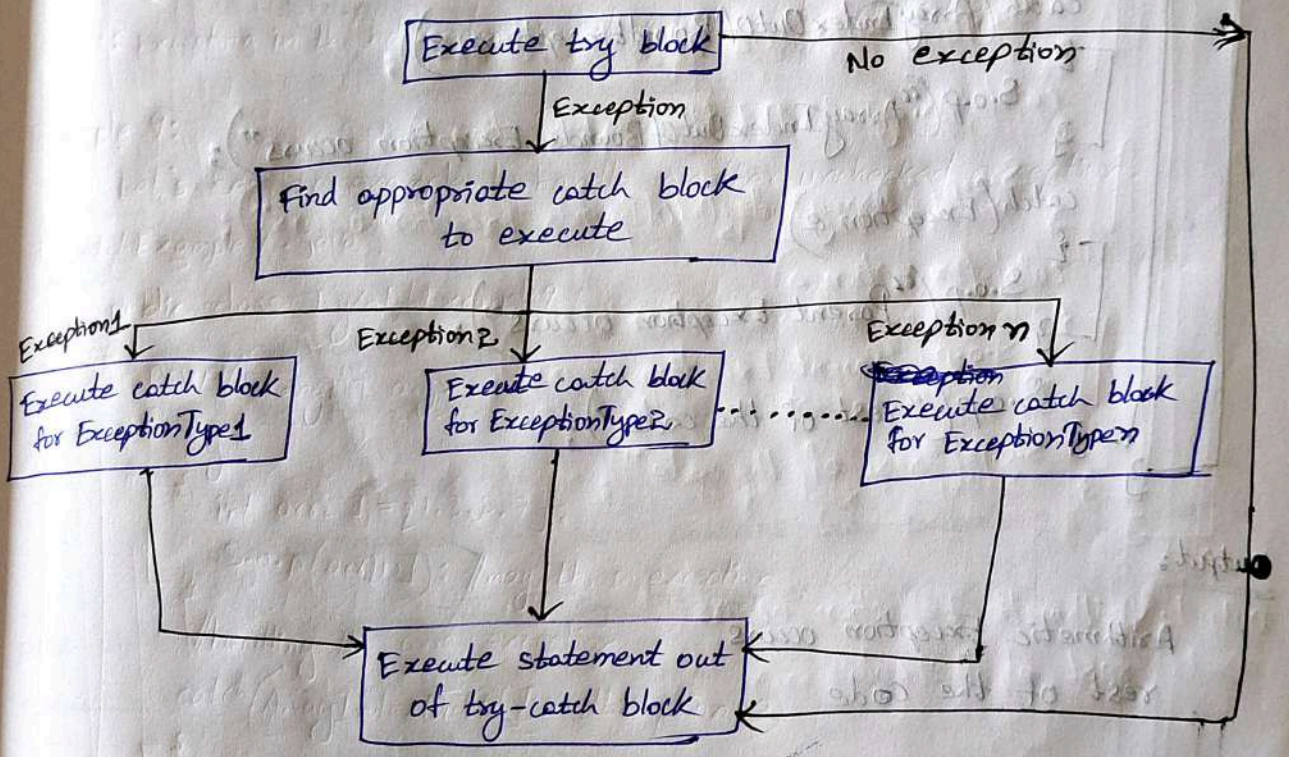
Java Multi-catch block:

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember:

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, ie., catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block:



Example 1:

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
public class MultipleCatchBlock1
{
    psvm(String[] args)
    {
    try
    {
        int a[] = new int[5];
        a[5] = 30/0;
    }
    catch(ArithmeticException e)
    {
        S.o.p("Arithmetic Exception occurs");
    }
    catch(ArrayIndexOutOfBoundException e)
    {
        S.o.p("ArrayIndexOutOfBounds Exception occurs");
    }
    catch(Exception e)
    {
        S.o.p("Parent Exception occurs");
    }
        S.o.p("rest of the code");
    }
}
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2:

MultipleCatchBlock2.java

```
public class MultipleCatchBlock2
{
    PSVM(String[] args)
    {
        try
        {
            int a[] = new int[5];
            S.o.p(a[10]);
        }
        catch(ArithmeticException e)
        {
            S.o.p("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            S.o.p("ArrayIndexOutOfBounds Exception occurs");
        }
        catch(Exception e)
        {
            S.o.p("Parent Exception occurs");
        }
        S.o.p("rest of the code");
    }
}
```

Output: ArrayIndexOutOfBounds Exception occurs
rest of the code

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

Example 3:

In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

MultipleCatchBlock3.java

```
public class MultipleCatchBlock3
{
    psvm(String[] args)
    {
        try
        {
            int a[] = new int[5];
            a[5] = 30/0;
            S.o.p(a[10]);
        }
        catch(ArithmeticException e)
        {
            S.o.p("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            S.o.p("ArrayIndexOutOfBounds Exception occurs");
        }
        catch(Exception e)
        {
            S.o.p("Parent Exception occurs");
        }
        S.o.p("rest of the code");
    }
}
```

Output: Arithmetic Exception occurs
rest of the code

Example 4:

In this example, we generate NullPointerException, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class Exception will invoked.

MultipleCatchBlock4.java

```
public class MultipleCatchBlock4
{
    psvm(String[] args)
    {
        try
        {
            String s = null;
            s.o.p(s.length());
        }
        catch (ArithmeticException e)
        {
            s.o.p("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            s.o.p("ArrayIndexOutOfBounds Exception occurs");
        }
        catch(Exception e)
        {
            s.o.p("Parent Exception occurs");
        }
        s.o.p("rest of the code");
    }
}
```

Output:

```
Parent Exception occurs
rest of the code
```

Example 5:

Let's see an example, to handle the exception without maintaining the order of exceptions (ie, from most specific to most general).

MultipleCatchBlocks.java

```
class MultipleCatchBlocks
{
    PSVM(String args[])
    {
        try
        {
            int a[] = new int[5];
            a[5] = 30/0;
        }
        catch(Exception e)
        {
            S.o.p("Common task completed");
        }
        catch(ArithmeticException e)
        {
            S.o.p("task1 is completed");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            S.o.p("task2 completed");
        }
        S.o.p("rest of the code");
    }
}
```

Output: Compile-time error

Nested try Statements:

Nested try block:

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the inner try block can be used to handle ArrayIndexOutOfBoundsException while the outer try block can handle the ArithmeticException (division by zero).

Why use nested try block:

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
//main try block
try
{   statement 1;
    statement 2;
    //try catch block within another try block
    try
    {   statement 3;
        statement 4;
        //try catch block within nested try block
        try
        {   statement 5;
            statement 6;
        }
        catch (Exception e2)
        {   //exception message
        }
    }
    catch (Exception e1)
    {   //exception message
    }
}
//catch block of parent (outer) try block
catch (Exception e3)
{   //exception message
}
....
```

Java Nested try Example:

Example 1:

Let's see an example where we place a try block within another try block for two different exceptions.

NestedTryBlock.java

```
public class NestedTryBlock
{
    psvm(String args[])
    {
        //outer try block
        try
        {
            //inner try block1
            try
            {
                S.o.p("going to divide by 0");
                int b=39/0;
            }
            //catch block of inner try block1
            catch(ArithmeticException e)
            {
                S.o.p(e);
            }
            //inner try block2
            try
            {
                int a[] = new int[5];
                //assigning the value out of array bounds
                a[5] = 4;
            }
            //catchblock of inner try block2
            catch(ArrayIndexOutOfBoundsException e)
            {
                S.o.p(e);
            }
            S.o.p("other statements");
        }
        //catch block of outer try block
        catch(Exception e)
        {
            S.o.p("handled the exception(outer catch)");
        }
        S.o.p("normal flow..");
    }
}
```

Output:

```
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statements
normal flow..
```

When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

If none of the catch block specified in the code is unable to handle the exception, then the Java runtime system will handle the exception. Then it displays the system generated message for that exception.

Example 2:

Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

NestedTryBlock.java

```
public class NestedTryBlock2
{
    PSVM(String args[])
    {
        //outer(main) try block
        try
        {
            //inner try block1
            try
            {
                //inner try block2
                try
                {
                    int arr[] = {1,2,3,4};
                    //printing the array element out of its bounds
                    S.o.p(arr[10]);
                }
                // to handles ArithmeticException
                catch(ArithmeticException e)
                {
                    S.o.p("Arithmetic exception");
                    S.o.p("inner try block2");
                }
            }
            // to handle ArithmeticException
            catch(ArithmeticException e)
            {
                S.o.p("Arithmetic exception");
                S.o.p("inner try block1");
            }
        }
        // to handle ArrayIndexOutOfBoundsException
        catch(ArrayIndexOutOfBoundsException e4)
        {
            S.o.print(e4);
            S.o.p("outer(main) try block");
        }
        catch(Exception e5)
        {
            S.o.print("Exception");
            S.o.p("handled in main try-block");
        }
    }
}
```

Output: java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 4 Outer(main) try block

finally:

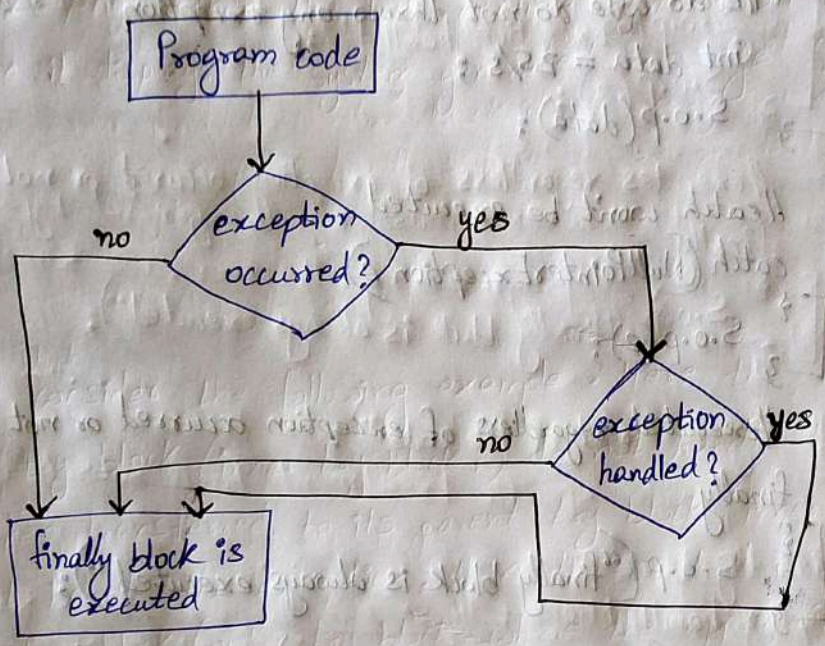
Java finally block:

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block:



Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

→ finally block in Java can be used to put "cleanup" code such as closing a file, closing connection, etc.

→ The important statements to be printed can be placed in the finally block.

Usage of Java finally:

Let's see the different cases where Java finally block can be used.

Case 1: When an exception does not occur

Let's see the below example where the Java program does not throw any exception, and the finally block is executed after the try block.

TestFinallyBlock.java

```
class TestFinallyBlock
{
    PSVM(String args[])
    {
        try
        {
            //below code do not throw any exception
            int data = 25/5;
            S.o.p(data);
        }
        //catch won't be executed
        catch (NullPointerException e)
        {
            S.o.p(e);
        }
        //executed regardless of exception occurred or not
        finally
        {
            S.o.p("finally block is always executed");
        }
        S.o.p("rest of the code...");
    }
}
```

Output: finally block is always executed
rest of the code...

Case 2: When an exception occur but not handled by the catch block

Let's see the following example. Here, the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

TestFinallyBlock1.java

```
public class TestFinallyBlock1
{
    PSVM(String args[])
    {
        try
        {
            S.o.p("Inside the try block");
            //below code throws divided by zero exception
            int data = 25/0;
            S.o.p(data);
        }
        //cannot handle Arithmetic type exception
        //can only accept Null Pointer type exception
        catch(NullPointerException e)
        {
            S.o.p(e);
        }
        //executes regardless of exception occured or not
        finally
        {
            S.o.p("finally block is always executed");
        }
        S.o.p("rest of the code...");
    }
}
```

output: Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero

Case 3: When an exception occurs and is handled by the catch block

Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

TestFinallyBlock2.java

```
public class TestFinallyBlock2
{
    PSVM(String args[])
    {
        try
        {
            S.o.p("Inside try block");
            //below code throws divide by zero exception
            int data = 25/0;
            S.o.p(data);
        }
        //handles the Arithmetic Exception / Divide by zero exception
        catch(ArithmeticException e)
        {
            S.o.p("Exception handled");
            S.o.p(e);
        }
        //executes regardless of exception occured or not
        finally
        {
            S.o.p("finally block is always executed");
        }
        S.o.p("rest of the code");
    }
}
```

Output: Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Creating own exception sub classes:

The Java programming language allow us to create our own exception classes which are basically subclasses built-in class Exception.

To create our own exception class simply create a class as a subclass of built-in Exception class.

We may create constructor in the user-defined exception class and pass a string to Exception class constructor using super(). We can use getMessage() method to access the string.

Let's look at the following Java code that illustrates the creation of user-defined exception.

Example 1:

```
import java.util.Scanner
class NotEligibleException extends Exception
{
    NotEligibleException(String msg)
    {
        super(msg);
    }
}
class VoterList
{
    int age;
    VoterList(int age)
    {
        this.age = age;
    }
    Void checkEligibility()
    {
        try
        {
            if(age < 18)
            {
                throw new NotEligibleException("Error: Not eligible for vote due to
                                                under age.");
            }
            S.o.p("Congrates! You are eligible for vote.");
        }
        catch(NotEligibleException nee)
        {
            S.o.p(nee.getMessage());
        }
    }
    PSVM(String args[])
    {
        Scanner input = new Scanner(System.in);
        S.o.p("Enter your age in years:");
        int age = input.nextInt();
        VoterList person = new VoterList(age);
        person.checkEligibility
    }
}
```

Output:

```
Enter your age in years:
16
Error: Not eligible for vote due to under age.
```

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which NotEligibleException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e., Exception class that can be obtained using getMessage() method on the object we have created.

Why use custom exceptions?

Java exception cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

Following are few of the reasons to use custom exceptions:

→ To catch and provide specific treatment to a subset of existing Java exceptions.

→ Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In Order to create custom exception, we need to extend Exception class that belongs to java.lang package.

Consider the following example, where we create a custom exception named WrongFileNameException:

```
public class WrongFileNameException extends Exception
{
    public WrongFileNameException(String errorMessage)
    {
        super(errorMessage);
    }
}
```

Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.

Example 2:

TestCustomException2.java

```
//class representing custom exception
class MyCustomException extends Exception
{
}

//class that uses custom exception MycustomException
public class TestCustomException2
{
    //main method
    PSVM(String args[])
    {
        try
        {
            //throw an object of user defined exception
            throw new MyCustomException();
        }
        catch(MyCustomException ex)
        {
            S.o.p("Caught the exception");
            S.o.p(ex.getMessage());
        }
        S.o.p("rest of the code...");
    }
}
```

output: Caught the exception
null
rest of the code...

Multithreading

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate seperate memory area so saves memory, and context-switching between the threads takes less time than process.

Advantages of Java Multithreading:

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
2) You can perform many operations together, so it saves time.
3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking:

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

→ Process-based Multitasking (Multiprocessing)
→ Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

→ Each process has an address in memory. In other words, each process allocates a seperate memory area.
→ A process is heavyweight.
→ Cost of communication between the process is high.
→ Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

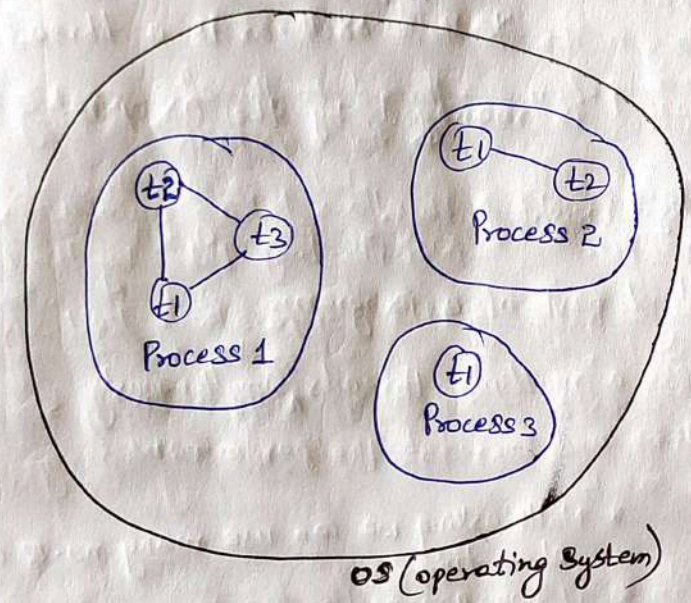
→ Threads share the same address space.
→ A thread is lightweight.
→ Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java:

A thread is a lightweight subprocess, the smallest unit of processing. It is a seperate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Java Thread class:

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Multi Tasking → we can able to perform multiple task simultaneously

Process Based
(Multi Processing)

→ Each task is performed seperate process
Ex: In PC (Personal Computer)
1) → we are doing program in notepad
2) → we are listening songs in VLC player
3) → we are downloading movie in utourrents
* These 3 above tasks are performed seperate process for each task. One task is doesn't link to the another task.

Thread Based
(Multi Threading)
Ex:- Programmatic level (PL)

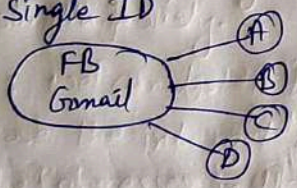
→ Each task is seperate independent part of a same program (or) process.
Ex: car racing game
→ Our car is seperate ~~process~~ thread
→ remaining individual cars perform seperate - seperate threads.
→ But finally all the cars belongs to the same game (or) process (or) Task.
* Here each car is a thread.

(Process Based & Thread Based is managed by the OS operating System)

Thread: A thread is a light weight Process.
(or)
Seperate independent part of a program.

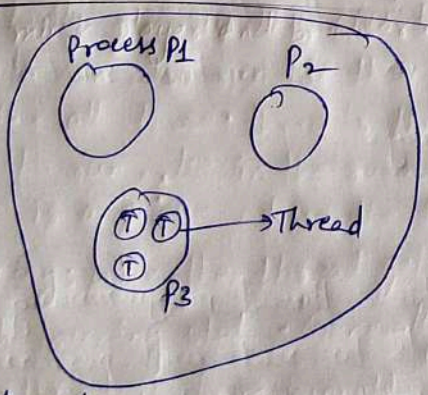
Multi-Threading: It is process a multiple threads works simultaneously
→ It reduces the response time and increase the performances.

Ex: Single ID



* Multitasking we can use many no. of clients at-a-time.
* If we use one of the other, then it takes lots of time. To overcome this we use multi-tasking.

Ex:



Multi-Threading:
↳ Process, it executes multiple threads simultaneously (or) sub-part of process (or) Lightweight process.

Differences between thread-based multitasking and process-based multitasking:

Multitasking operating system is an operating system that gives you the perception of two or more tasks/jobs/processes running simultaneously. It does this by dividing system resources amongst these tasks/jobs/processes and switching between the tasks/jobs/processes while they are executing over and over again. The CPU processes only one task at a time, but in Multitasking the switching is so fast that it looks like the CPU is executing multiple processes simultaneously. They can support either preemptive multitasking, where the OS doles out time to applications (Virtually all modern OSes) or cooperative multitasking, where the OS waits for the program to give back control (Windows 3.x, Mac OS 9 and earlier), leading to hangs and crashes. Also known as Timesharing, multitasking is a logical extension of multiprogramming.

Multitasking Programming has Two Types:

1) Process-based Multitasking
2) Thread-based Multitasking

Process-Based Multitasking

→ In process-based multitasking, two or more processes and programs can be run concurrently.

→ In process-based multitasking, a process or a program is the smallest unit.

→ The program is a bigger unit.

→ Process-based multitasking requires more overhead.

→ The process requires its own address space.

→ The process to process communication is expensive.

→ Here, it is unable to gain access over the idle time of the CPU.

→ It is a comparatively heavyweight.

→ It has a faster data rate for multi-tasking because two or more processes/programs can be run simultaneously.

Example: we can listen to music and browse the internet at the same time. The processes in this example are the music player and browser.

Thread-Based Multitasking

→ In thread-based multitasking, two or more threads can be run concurrently.

→ In thread-based multitasking, a thread is the smallest unit.

→ Thread is a smaller unit.

→ Threads ~~share the same~~ -based multitasking, requires less overhead.

→ Thread share the same address space.

→ Thread to Thread communication is not expensive.

→ It allows taking gain access over idle time taken by the CPU.

→ It is comparatively lightweight.

→ It has a comparatively slower data rate multi-tasking.

Example: Using a browser we can navigate through the webpage and at the same time download a file. In this example, navigation is one thread and downloading is another thread. Also in a word-processing like Ms Word, we can type ~~text~~ text in one thread, and spell-checker checks for mistakes in another thread.

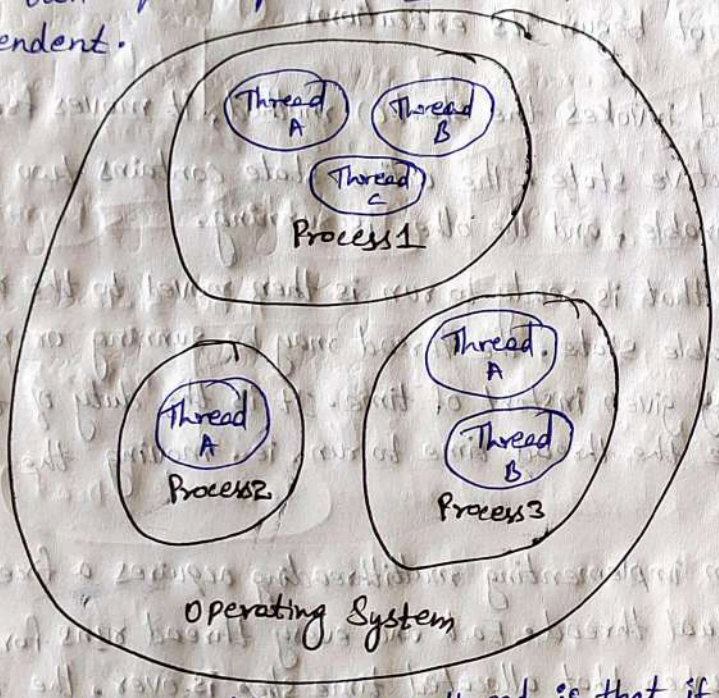
Java Thread Model: (or) Life cycle of thread (or) A Thread Model:

↳ It can be handled by "JVM"

Before introducing the thread concept, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, Thread concept was introduced.

A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the Thread concept in Java. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own seperate path for execution, so each thread of a process is independent.



Another benefit of using thread is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as Multithreading.

In a simple way, a Thread is a:

→ Feature through which we can perform multiple activities within a single process.
→ Lightweight process.
→ Series of executed statements.
→ Nested sequences of method calls.

Thread Model (or) Life cycle of a Thread (or) Thread states:

In Java, a thread always exists in any one of the following states. These states are:

1) New
2) Active
3) Blocked / Waiting
4) Timed Waiting
5) Terminated

Explanation of Different Thread States:

1) New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

2) Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: One is runnable, and the other is running.

→ Runnable: A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, ie., moving the thread the running state.

A program implementing multithreading acquires a fixed slice of time to each individual threads. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.

→ Running: When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

3) Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child thread complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

4) Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (it name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, ~~a timed waiting~~ another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

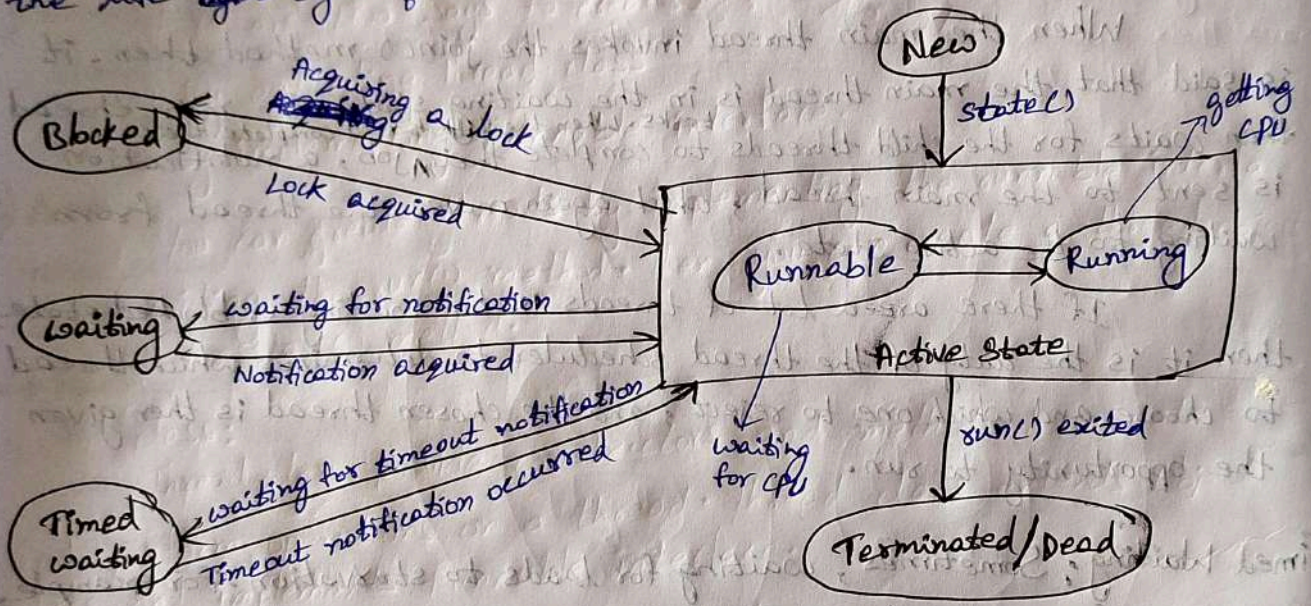
5) Terminated: A thread reaches the termination state because of the following reasons:

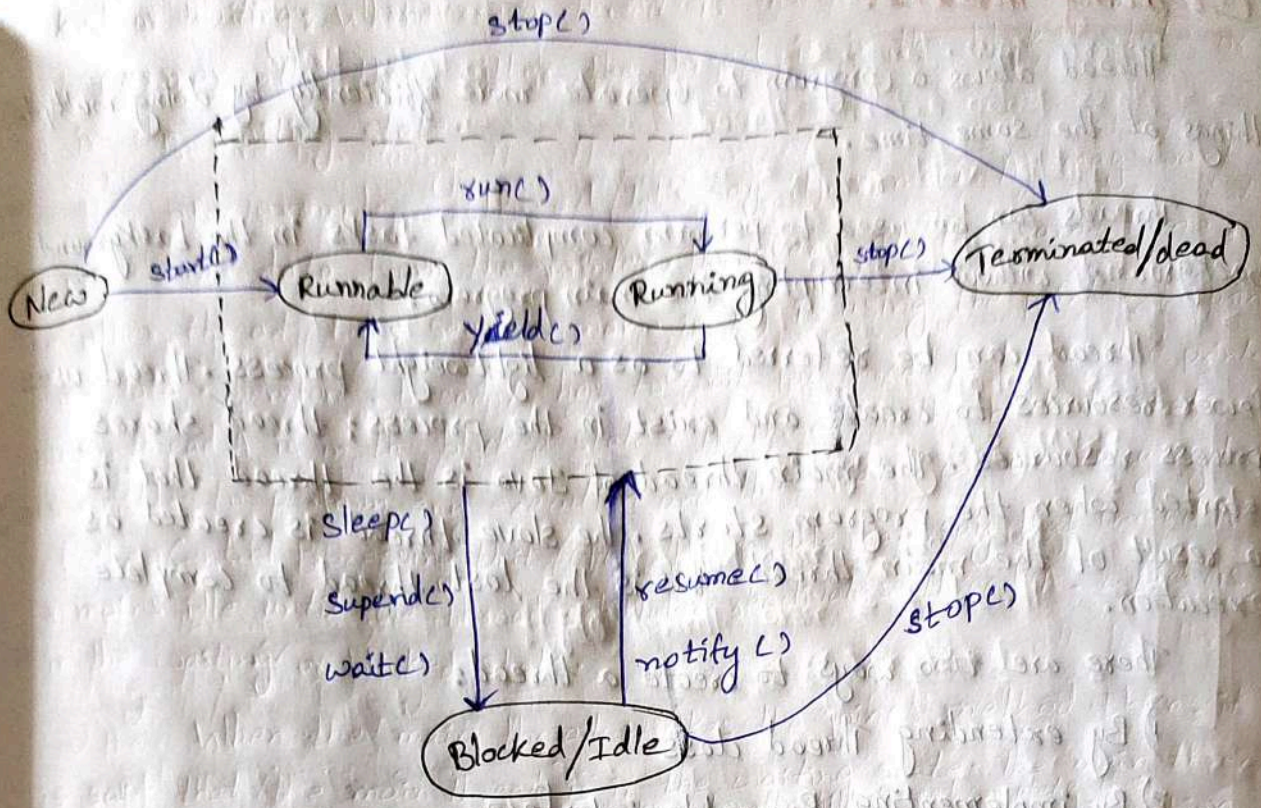
→ When a thread has finished its job, then it exists or terminates normally.

→ Abnormal termination: It occurs when some unusal events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after Kill) the dead thread.

The following diagram shows the different states involved in the life cycle of a thread.





1) New → Thread t = new Thread()

2) Runnable → t.start()

3) Running → run() will executing

4) Blocked/Idle → sleep(), suspend(), wait()

5) Terminated/dead → t.stop()

resume()

notify()

Thread is going to be in waiting state if notify() comes then it will execute

Thread is going to sleep for some time
Thread.sleep(2000);
↓
milliseconds

Thread is going to be suspended state
block
unblock
when resume() method comes then it is going to execute.

Creating Threads:

Thread allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the ~~background~~ background without interrupting the main program.

Thread can be referred to as a lightweight process. Thread uses fewer resources to create and exist in the process; thread shares process resources. The main thread of Java is the thread that is started when the program starts. The slave thread is created as a result of the main thread. This is the last thread to complete execution.

There are two ways to create a thread:

1) By extending Thread class
2) By implementing Runnable interface.

You can create threads by implementing the runnable interface and overriding the run() method. Then, you can create a thread object and call the start() method.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

```
// Start a newly created thread.
// Thread moves from new state to runnable state
// When it gets a chance, executes the target run() method
  public void start()
```

Runnable interface:

Any class with instances that are intended to be executed by a thread should implement the Runnable interface. The Runnable interface has only one method, which is called run().

```
// Thread action is performed
  public void run()
```

Benefits of creating threads:

→ When compared to processes, Java Threads are more lightweight; it takes less time and resources to create a thread.
→ Threads share the data and code of their parent process.
→ Thread communication is simpler than process communication.
→ Context switching between threads is usually cheaper than switching between processes.

Calling run() instead of start():

The common mistake is starting a thread using run() instead of start() method.

```
Thread myThread = new Thread (MyRunnable())
  myThread.run(); //should be start();
```

The run() method is not called by the thread you created. Instead, it is called by the thread that created the myThread.

Thread:

→ A Thread is a process or execution of group of statements
→ A Thread is a sequence of executing instructions which can run independently of other threads.

To create a thread we can follow below step.

step 1: To create a class that extends thread class or Runnable interface.

Ex: class Robo extends Thread

(or)

class Robo implements Runnable

step 2: write run() method with body.

step 3: create an object to the class and attach to a thread

Ex: Robo obj1 = new Robo();
Thread t1 = new Thread(obj1);

step 4: start the thread using start() method.

Ex: t1.start();

→ Program:

```
import java.lang.*;
class Robo extends Thread
{
    public void run()
    {
        S.o.p("Hello");
    }
};
class Mrobo
{
    PSVM(String args[])
    {
        Robo obj = new Robo();
        Thread t = new Thread(obj);
        t.start();
    }
}
```

output: Hello

Example 1: By using Thread class

```
import java.io.*;
class Demo extends Thread
{
    public void run()
    {
        S.o.p("Welcome Konica");
    }
    PSVM(String args[])
    {
        Demo d = new Demo(); // creating thread
        d.start(); // starting thread
    }
}
```

output: Welcome Konica

Example 2: By Implementing Runnable interface

```
import java.io.*;
class Demo implements Runnable
{
    PSVM(String args[])
    {
        //create an object of Runnable target
        Demo d = new Demo();

        //pass the runnable reference to Thread
        Thread t = new Thread(d, "Konica");

        //start the thread
        t.start();

        //get the name of the thread
        S.o.p(t.getName());
    }
    public void run()
    {
        S.o.p("Inside run method");
    }
}
```

output: Konica
Inside run method

Thread Methods:

→ start(): This method is used to execution of a thread. (JVM calls the run() method on the thread)

Ex: Thread t = new Thread();
t.start();

→ run(): It is used to perform action for a thread.

Ex: public void run()
{
// code
}

Same

→ getName(): This method is used to get Name of thread.

Ex: S.o.p(t.getName()); // o/p: Thread 0. (Thread name starts with 0 (0, 1, 2, 3, ... etc))

→ setName(): This method is used to set Name of a existing thread.

Ex: t.setName("robo");

→ getPriority(): This method is used to get Priority of a thread.

Ex: MinPriority = 1
NormalPriority = 5 → default
Max Priority = 10

→ setPriority(): This method is used to set Priority of a existing thread

Ex: t.setPriority(10);
S.o.p(t.getPriority());

→ sleep(milliseconds): It is a static method (To block thread upto certain time. If the time limit complete then automatically it restarts the thread.)

Ex: Thread.sleep(2000);

→ suspend(): (To block the thread upto resume() method comes)

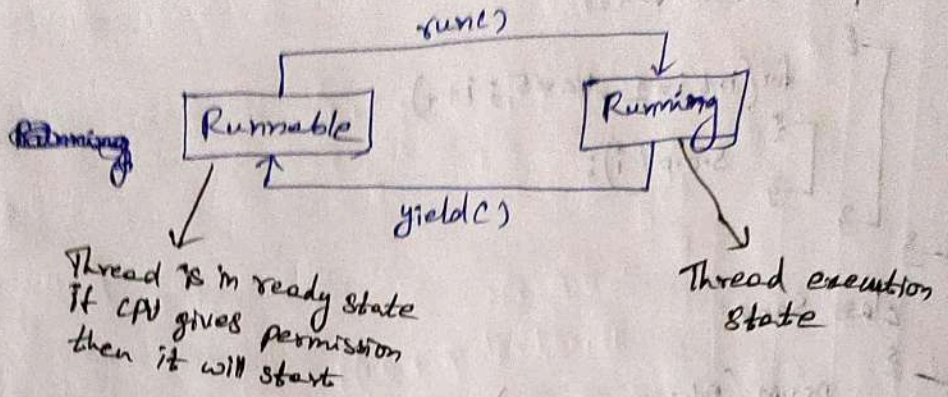
→ resume(): ~~(It restarts the 0~~ restarts (It restarts the suspended thread)

→ wait(): (To block or waiting state of all the thread).

→ notify(): (Only first thread will restart in the waiting threads and remaining threads will be in waiting state.)

→ notifyAll(): (All the threads will be restart when the threads are in the waiting state.)

→ yield(): Running state to Runnable state
execution state to ready state



→ run(): Runnable state to Running state
ready state to execution state.

→ join(): If first thread executes completely then only next thread is going to execute. (one thread after the another thread)
(waits for a thread to die)

Ex: t1.start();
t1.join(); → after completion of t1 thread then only it will start executing t2 thread.
t2.start();

→ stop(): After execution, the thread will be closed

→ isAlive(): To know the status of the thread ie, thread is in running state or not.
true → Thread is running state
false → Thread is not in running state

→ Program: (getName(), setName(), getPriority(), setPriority())

```
class Chitti extends Thread
{
    public void run()
    {
        for(int i=1 ; i<=5 ; i++)
        {
            S.o.p(i);
        }
    }
}
class Sana
{
    PSVM(String args[])
    {
        Chitti t1 = new Chitti();
        S.o.p("T1 thread name = " + t1.getName());
        t1.setName("Chitti");
        S.o.p("T1 thread name after change = " t1.getName());
        S.o.p("T1 thread priority = " + t1.getPriority());
        t1.setPriority(10);
        S.o.p("T1 thread priority after change = " t1.getPriority());
        t1.start();
    }
}
```

Output:

```
T1 thread name = Thread-0
T1 thread name after change = Chitti
T1 thread priority = 5
T1 thread priority after change = 10
1
2
3
4
5
```

→ Program: (isAlive())

```
class Chitti extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            try
            {
                S.o.p(i);
                Thread.sleep(1000);
            }
            catch(Exception e)
            {
            }
        }
    }

class Sana
{
    psvm(String args[])
    {
        Chitti t1 = new Chitti();
        S.o.p("thread status" + t1.isAlive());
        t1.start();
        S.o.p("thread status" + t1.isAlive());
    }
}
```

Output: thread status false
thread status true
1 → after 1 it takes 1000 milliseconds to print value 2.
2 (one by one it takes 1000 milliseconds to print another value).
3
4
5

Checking the thread is in running state or not by using isAlive() method.

Thread true → isAlive (running state)

Thread false → not isAlive (not running state)

→ **Program:** (Two threads running at-a-time)

```
class Chitti extends Thread
{
    public void run()
    {
        for(int i=1 ; i<=5 ; i++)
        {
            try
            {
                S.o.p(i);
                Thread.sleep(1000);
            }
            catch(Exception e)
            {
            }
        }
    }

Class Sana
{
    psvm(String args[])
    {
        Chitti t1 = new Chitti();
        Chitti t2 = new Chitti();
        t1.start();
        t2.start();
    }
}
```

Output:

1 → t1 execute & sleep 1000 milliseconds
1 → t2 execute & slep 1000 milliseconds
2 → t1 "
2 → t2 "
3 → t1 "
3 → t2 "
4 →
4 →
5 →
5 →

→ **Program:** (After completion of one thread only it will execute the next thread by using join() method.)

```
class Chitti extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            try
            {
                S.o.p(i);
                Thread.sleep(1000);
            }
            catch(Exception e)
            {
            }
        }
    }
class Sana
{
    PSVM(String args[])
    {
        Chitti t1 = new Chitti();
        Chitti t2 = new Chitti();
        t1.start();
        try
        {
            t1.join();
        }
        catch(Exception e)
        {
        }
        t2.start();
    }
}
```

Output:

1 2 3 4 5 } t1 thread complete execution & sleep 1000 milliseconds each iterator

1 2 3 4 5 } t2 thread execution & Sleep 1000 milliseconds each iterator

(After completion of t1 thread only than t2 thread will execute. One thread after another thread)

Commonly used Constructors of Thread class:

Thread()
Thread(String name)
Thread(Runnable r)
Thread(Runnable r, String name)

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

Starting a thread:

The start() method of Thread class is used to start a newly created thread. It performs the following tasks:

→ A new thread starts (with new callstack).
→ The thread moves from New state to the Runnable state.
→ When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class:

```
class Multi extends Thread
{
  public void run()
  {
    S.o.p("thread is running...");
  }
  psvm(String args[])
  {
    Multi t1 = new Multi();
    t1.start();
  }
}
```

Output:

thread is running...

2) Java Thread Example by implementing Runnable interface:

```
class Multi implements Runnable
{
    public void run()
    {
        S.o.p("thread is running...");
    }
    PSVM(String args[])
    {
        Multi m1 = new Multi();
        Thread t1 = new Thread(m1); // Using the constructor Thread(Runnable r)
        t1.start();
    }
}
```

Output:

thread is running...

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create the Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

3) Using the Thread class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors.

```
public class MyThread
{
    PSVM(String args[])
    {
        //Creating an object of the Thread class using the Constructor
            Thread(String Name)
        Thread t = new Thread("My first thread");

        //the start() method moves the thread to the active state
        t.start();

        //getting the thread name by invoking the getName() method
        String str = t.getName();
        S.o.p(str);
    }
}
```

Output:

My first thread

4) Using the Thread Class: Thread (Runnable r, String name)

```
public class MyThread implements Runnable
{
    public void run()
    {
        S.o.p("Now the thread is running...");
    }

    PSVM(String args[])
    {
        //creating an object of the class MyThread
        Runnable r1 = new MyThread();

        //creating an object of the class Thread using
            Thread(Runnable r, String name)
        Thread t1 = new Thread(r1, "My new thread");

        // the start() method moves the thread to the active state
        t1.start();

        //getting the thread name by invoking the getName() method
        String str = t1.getName();
        S.o.p(str);
    }
}
```

Output:

```
My new thread
Now the thread is running...
```

Thread Priorities:

As we already know java being completely object oriented works within a multithreading environment in which thread scheduler assigns the processor to a thread based on the priority of thread. Whenever we create a thread in java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

In a java programming language, every thread has a property called priority. Most of the scheduling algorithms use the thread priority to schedule the execution sequence. In java, the thread priority range from 1 to 10. Priority 1 is considered as the lowest priority and priority 10 is considered as the highest priority. The thread with more priority allocates the processor first.

The java programming language Thread class provides two methods setPriority(int) and getPriority() to handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

MAX_PRIORITY - It has the value 10 and indicates highest priority.
NORM_PRIORITY - It has the value 5 and indicates normal priority.
MIN_PRIORITY - It has the value 1 and indicates lowest priority.

→ The default priority of any thread is 5 (ie, NORM_PRIORITY).

setPriority() method:

The setPriority() method of Thread class used to set the priority of a thread. It takes an integer range from 1 to 10 as an argument and returns nothing (void).

Ex: threadObject.setPriority(4);
(or)
threadObject.setPriority(MAX_PRIORITY);

getPriority() method:

The getPriority() method of Thread class used to access the priority of a thread. It does not takes any argument and returns name of the thread as String.

Ex: String threadName = threadObject.getPriority();

→ Program:

```
class SampleThread extends Thread
{
    public void run()
    {
        S.o.p("Inside SampleThread");
        S.o.p("Current Thread: " + Thread.currentThread().
                                    getName());
    }
}

public class My_Thread_Test
{
    PSVM(String args[])
    {
        SampleThread ot1 = new SampleThread();
        SampleThread ot2 = new SampleThread();
        ot1.setName("first");
        ot2.setName("second");

        ot1.setPriority(4);
        ot2.setPriority(Thread.MAX_PRIORITY);

        ot1.start();
        ot2.start();
    }
}
```

output:
```
Inside SampleThread
Current Thread: second
Inside SampleThread
Current Thread: first
```

→ In java, it is not guaranteed that threads executed according to their priority because it depends on JVM specification that which scheduling it chooses.

Creating Multiple Threads:

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms:

1) Extending the Thread class
2) Implementing the Runnable interface

Thread creating by extending the Thread class:

We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

→ Program:

```
// Java code for thread creation by extending the Thread class.
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            S.o.p("Thread " + Thread.currentThread().getId() +
                                    " is running");
        }
        catch(Exception e)
        {
            // Throwing an exception
            S.o.p("Exception is caught");
        }
    }
}
```

```
// Main Class
public class Multithread
{
    PSVM (String[] args)
    {
        int n=8; // Number of threads
        for(int i=0; i<n; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}
```

output:

```
Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running
```

Thread creating by implementing the Runnable Interface:

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

→ Program:

```
// Java code for thread creation by implementing the Runnable Interface
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            S.o.p("Thread " + Thread.currentThread().getId() +
                                " is running");
        }
        catch (Exception e)
        {
            // Throwing an exception
            S.o.p("Exception is caught");
        }
    }
}

// Main class
class Multithread
{
    PSVM(String[] args)
    {
        int n = 8; // Number of threads
        for(int i=0; i<n; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Output:

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

Thread class vs Runnable Interface:

1) If we extends the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

2) We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

3) Using runnable will give you an object that can be shared amongst multiple threads.

→ **Program:**

```
import java.lang.*;
class Robo extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            S.o.p(i);
        }
    }

class Mrobo
{
    PSVM(String args[])
    {
        Robo O1 = new Robo();
        Robo O2 = new Robo();
        Thread t1 = new Thread(O1);
        Thread t2 = new Thread(O2);
        t1.start();
        t2.start();
    }
}
```

output:

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

Synchronizing Threads:

Synchronization in Java:

Multi-threaded program may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

Synchronization in java is the capability to control the access of multiple threads to any shared resource. In the Multithreading concept, multiple threads try to access the shared resources at a time to produce inconsistent results. The synchronization is necessary for reliable communication between threads.

Why we use synchronization:

→ Synchronization helps in preventing thread interference.

→ Synchronization helps to prevent concurrency problems.

Java synchronization is used to make sure by some synchronization method that only one thread can access the resource at a given point in time.

Types of Synchronization:

Synchronization is classified into two types

→ Process Synchronization

→ Thread Synchronization

→ Process Synchronization:

The process is nothing but a program under execution. It runs independently isolated from another process. The resources like memory and CPU time, etc. are allocated to the process by the operating system.

→ Thread Synchronization:

Thread synchronization is two types, they are:

1) Mutual Exclusive:

A Mutex or Mutual Exclusive helps only one thread to access the shared resources. It won't allow the accessing of shared resources at a time. It can be achieved in the following ways:

→ Synchronized Method
→ synchronized block
→ static synchronization

2) Cooperation (Inter Thread communication)

Inter Thread communication or cooperation is a communication of two or more threads with each other. It can be done by using the following methods.

→ wait()
→ notify()
→ notifyAll()

Lock Concept in Java:

Synchronization mechanism developed by using the synchronized keyword in Java language. It is built on top of the locking mechanism, this locking mechanism is taken care by Java Virtual Machine (JVM). The synchronized keyword is only applicable for methods and blocks, it can't apply to classes and variables.

Synchronized keyword in java creates a block of code is known as a critical section. To enter into the critical section thread needs to obtain the corresponding object's lock.

Unit-5

User Interface Design:

The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface:

There are two main types of User Interface:

→ Text-Based User Interface or command Line Interface.

→ Graphical User Interface (GUI)

Text-Based User Interface (or) Command Line User Interface (CUI):

This method relies primarily on the keyboard. A typical example of this is Unix.

Advantages:

→ Many and easier to customizations options.

→ Typically capable of more important tasks.

Disadvantages:

→ Relies heavily on recall rather than recognition.

→ Navigation is often more difficult.

Graphical User Interface (GUI):

GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the windows operating systems.

GUI characteristics:

Windows → Multiple windows allow different information to be displayed simultaneously on the user's screen.

Icons → Icons different types of information. On some systems, icons represent files. On other icons describes processes.

Menus → Commands are selected from a menu rather than typed in a command language.

Pointing → A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interests in a windows.

Graphics → Graphics elements can be mixed with text or the same display.

Advantages:

→ Less expert knowledge is required to use it.

→ Easier to Navigate and can look through folders quickly in a guess and check manner.

→ The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages:

→ Typically decreased options.

→ Usually less customizable. Not easy to use one button for tons of different variations.

User Interface → A software that provides an interface between the user and the hardware.

- GUI — Graphical User Interface
- CUI — Command Line User Interface

Difference between GUI and CUI operating systems:

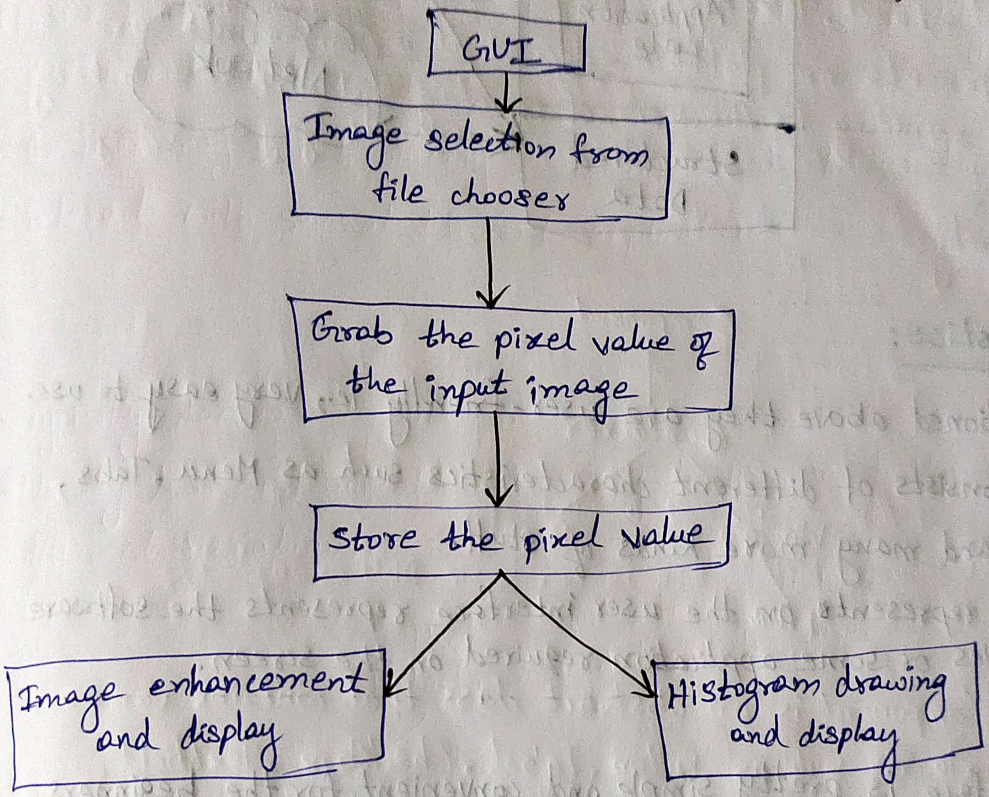
Terms	GUI (Graphical User Interface)	CUI (Command User Interface)
Interaction	A user interacts with the computer using Graphics like images, icons.	A user interacts with a computer using commands like text.
Navigation	Navigation is easy.	Navigation is difficult.
Precision	GUI has low precision.	CUI has high precision.
Usage	GUI is easy to use.	CUI is difficult to use and requires expertise.
Speed	GUI is of low speed.	CUI is of high speed.
peripherals used	GUI operating system requires a keyboard, mouse or any other pointing device.	CUI operating system requires the only keyboard.
Memory requirement	It required high memory.	It required low memory.
Flexibility	Highly flexible user interface.	Less flexible user interface.
customize	GUI is highly customizable	CUI appearance is not easily changeable.

GUI Introduction:

GUI stands for Graphical User Interface.

GUI is a user-friendly interface used to communicate with the help of electronic devices. It displays all the contents whether a text file or an object or pictures or videos and all the things that a user want to visualize. It interacts well and can be used whether everywhere whether a mobile phone, tablet, laptops, Personal Computer and all the other electronic devices. It can be used best in the gaming side where the resolution is being considered.

The basic GUI format is represented in the form of the diagram:

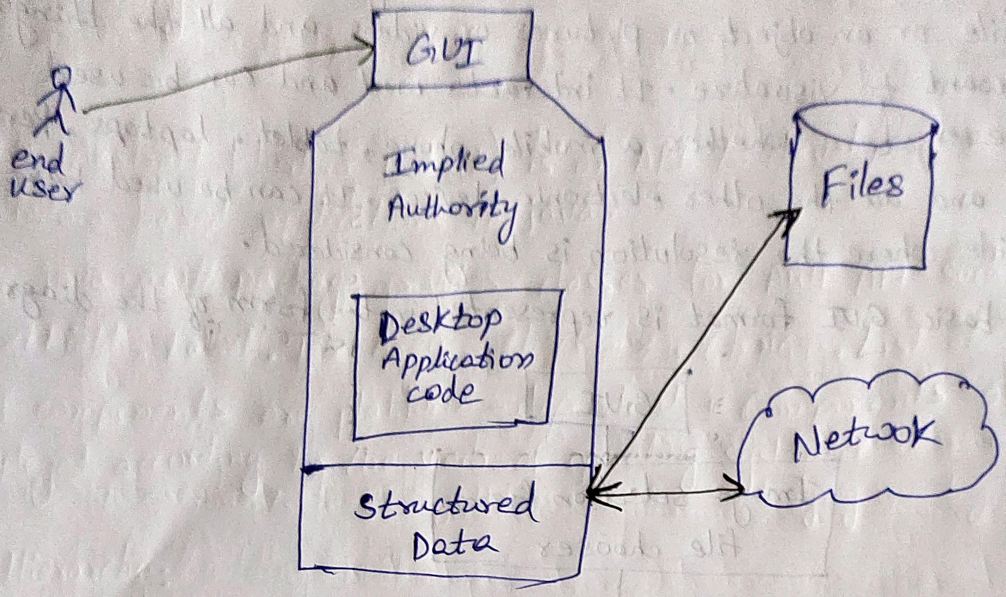


History of GUI:

Earlier, there was no GUI so people used to interact with the command-line interface (CLI). The CLI was not that friendly to use and the end-user was not familiar with all the commands. So to bridge this gap, GUI was introduced. The main aim of the GUI was to make the applications mush more user-friendly. People love when the task which they want to perform gets done easily and in an efficient manner.

GUI stresses one of the most important aspects which is "easy to use".

The basic structure of GUI using implied authority is given by the following diagram.



Characteristics:

→ As mentioned above they are user-friendly ie., very easy to use.

→ A GUI consists of different characteristics such as Menu, Tabs, Pointers and many more kinds of stuff.

→ The icons represents on the user interface represents the software or the files or some application required on the screen.

Advantages:

→ User Interface is pretty simple and convenient for the beginners to understand.

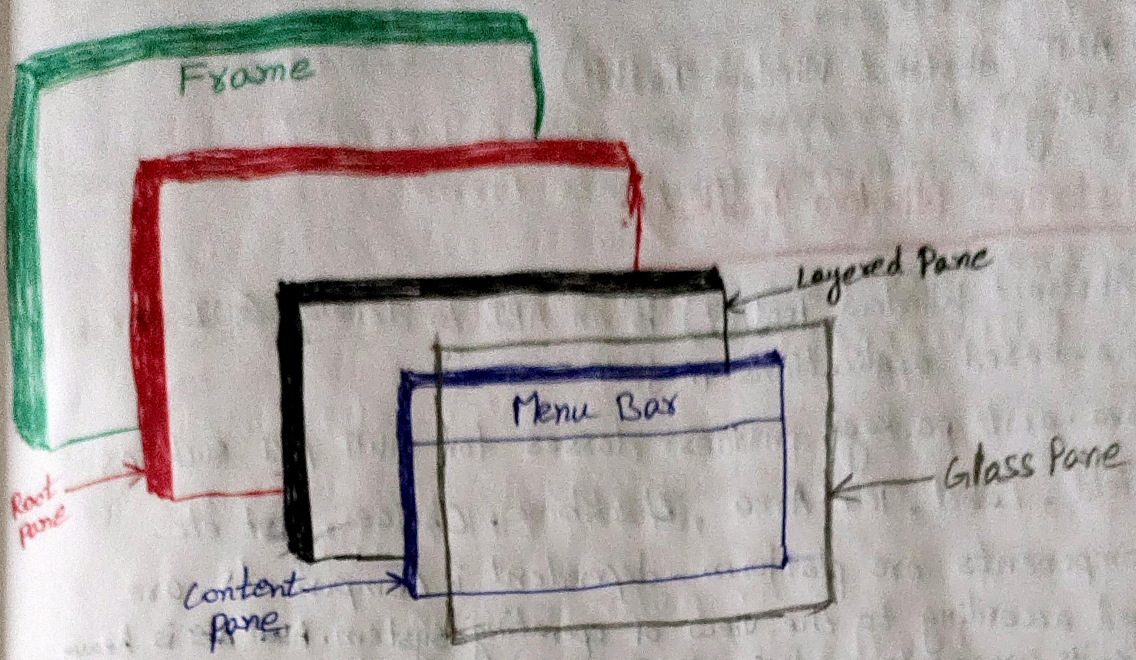
→ It is very intuitive and user friendly friendly and can be used by anyone.

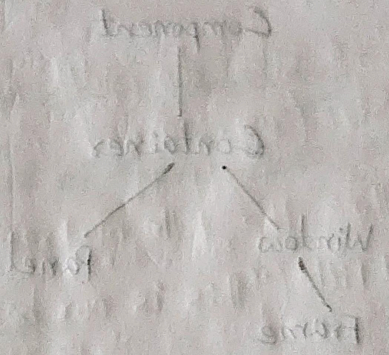
→ End-user need not memorize commands to perform actions in the application.

Disadvantages:

→ A bad GUI always create a problem for the gamer's as it creates a bad impact for them.

→ A GUI which is not user-friendly can mislead the user and the efficiency of completing the work reduces.


Frame
Layered Pane
Menu Bar
Glass Pane
Root Pane
Content Pane



GUI Programming:

Java Provides 2 Frameworks for building GUI - based applications. Those are

→ AWT (Abstract Window ToolKit)

→ Swing

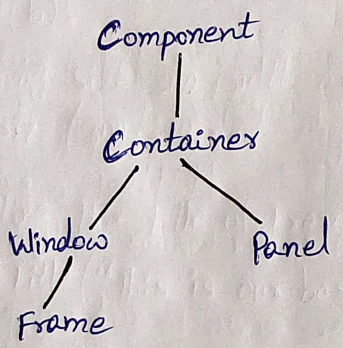
AWT (Abstract Window ToolKit):

→ AWT (Abstract Window ToolKit) is an API to develop GUI or Window-based applications in java.

→ The java.awt package provides classes for AWT API such as TextField, Label, TextArea, Checkbox, Choice, List etc.

→ AWT components are platform-dependent ie. Components are displayed according to the view of operating system. AWT is heavy weight ie, its components are using the resources of underlying operating system (os).

AWT Hierarchy:



Component: Component is an abstract class that contains various classes such as Button, Label, Checkbox, Textfield, Menu and etc. In order to place every component in a particular position on a screen, we need to add them to a container.

Container: The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The Container class extends Frame, Dialog, Window and Panel. It is basically a screen where the components are placed at their specific locations. Thus it contains and control the layout of components.

Window: The window is the container that have no borders and menu bar. You must use frame for creating a window.

Frame: The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield, scrollbar etc.

Panel: The Panel is the container that doesn't contain title bar, border and menu bars. It have other components like button, textfields etc.

Commonly used Methods of Component class:

Method	Description
add(Component c)	inserts a component on this component.
setSize(int width, int height)	sets the size (width and height) of the component.
setLayout(LayoutManager m)	defines the layout manager for the component.
setVisible(boolean status)	changes the visibility of the component, by default false.

AWT Examples:

To create simple AWT example, you need a frame. There are two ways to create a GUI using Frame in AWT.

1) By extending Frame class (inheritance)
2) By creating the object of Frame class (association)

1) AWT Example by Inheritance:

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

AWTExample1.java

```
//importing Java AWT class
import java.awt.*;

//extending Frame class to our class AWTExample1
public class AWTExample1 extends Frame
{
    //initializing using constructor
    AWTExample1()
    {
        //creating a button
        Button b = new Button("Click Me!!");

        //setting button position on screen
        b.setBounds(30,100,80,30);

        //adding button into frame
        add(b);

        //frame size 300 width and 300 height
        setSize(300,300);

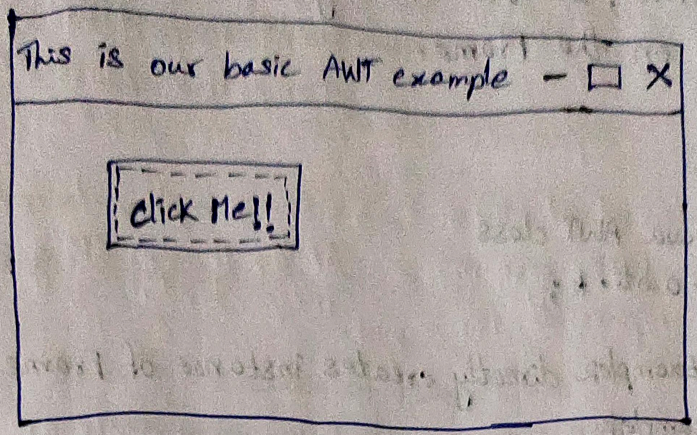
        //setting the title of Frame
        setTitle("This is our basic AWT example");

        //no layout manager
        setLayout(null);

        //now frame will be visible, by default it is not visible
        setVisible(true);
    }

    //main method
    PSVM(String args[])
    {
        //creating instance of Frame class
        AWTExample1 f = new AWTExample1();
    }
}
```

Output:



→ The setBounds(int x-axis, int y-axis, int width, int height) method is used in the above example that sets the position of the awt button.

2) AWT Example by Association:

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are creating a TextField, Label and Button component on the Frame.

AWTExample2.java

```
//importing Java AWT class
import java.awt.*;

//class AWTExample2 directly creates instance of Frame class
class AWTExample2
{
    //initializing using constructor
    AWTExample2()
    {
        //creating a Frame
        Frame f = new Frame();

        //creating a Label
        Label l = new Label("Employee id:");

        //creating a Button
        Button b = new Button("Submit");

        //creating a Textfield
        TextField t = new TextField();

        //setting position of above components in the frame
        l.setBounds(20,80,80,30);
        t.setBounds(20,100,80,30);
        b.setBounds(100,100,80,30);

        //creating a Textfield
        //adding components into frame
        f.add(b);
        f.add(l);
        f.add(t);

        //frame size 400 width and 300 height
        f.setSize(400,300);

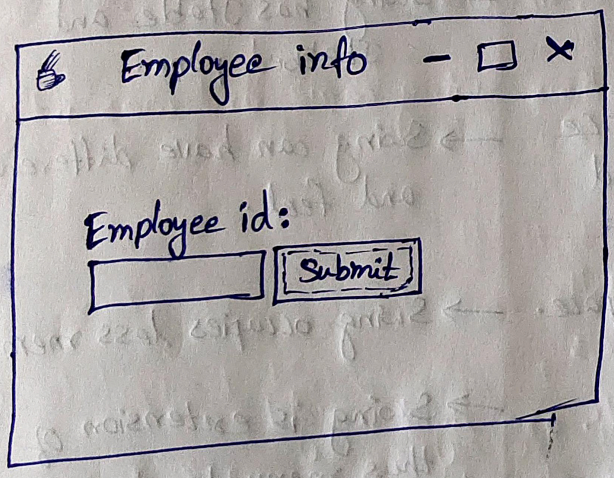
        //setting the title of frame
        f.setTitle("Employee info");
```

```
    //no layout
    f.setLayout(null);

    //setting visibility of frame
    f.setVisible(true);
}

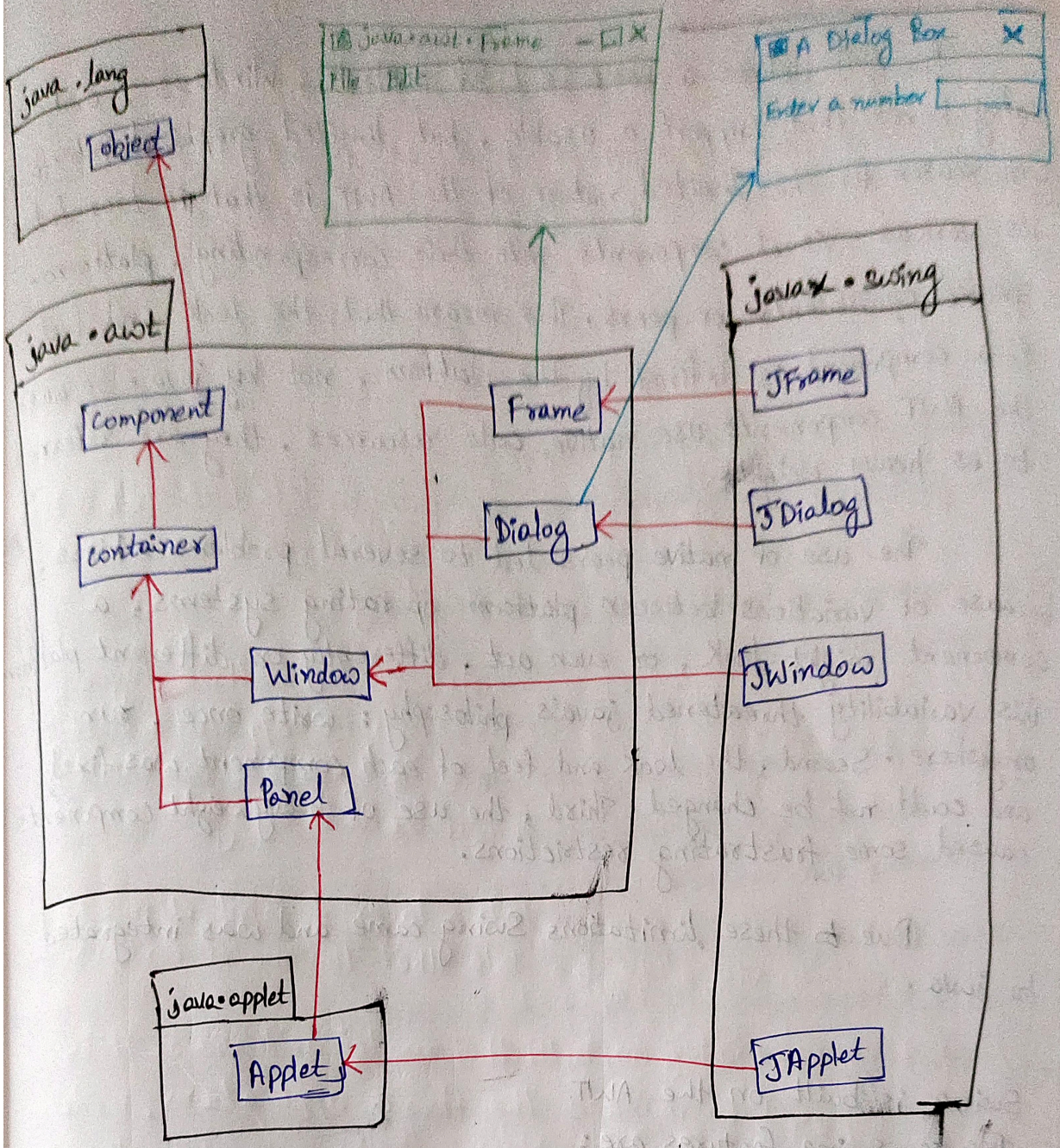
//main method
psvm(String args[])
{
    //creating instance of Frame class
    AWTExample2 awt_obj = new AWTExample2();
}
}
```

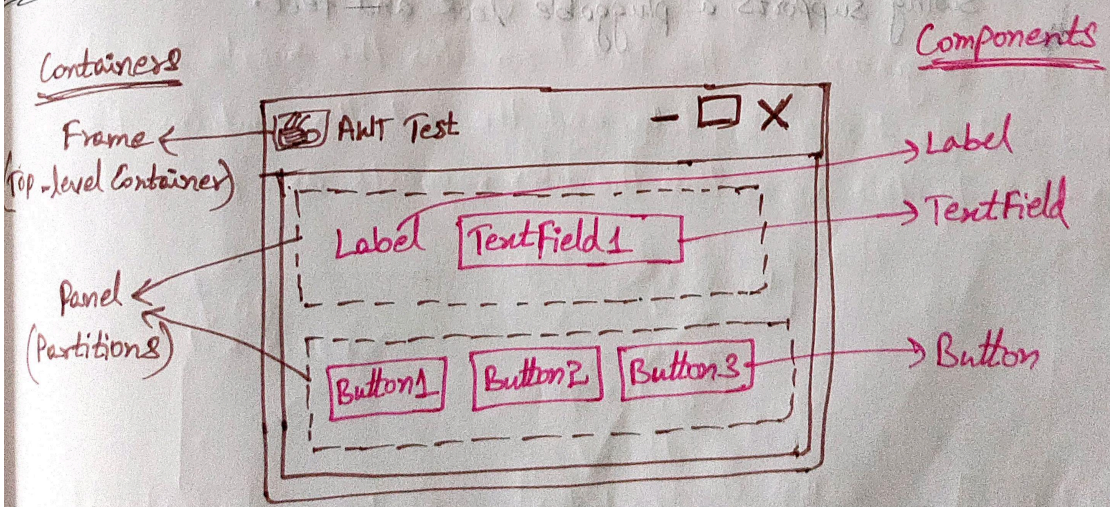
Output:



Difference between AWT and Swing:

AWT	Swing
→ AWT stands for Abstract Window Toolkit	→ Swing is part of Java Foundation Classes (JFC).
→ AWT is platform dependent. Its code resides in os of the system hence makes it impossible for platform independence.	→ Swing components are made in JAVA. Thus, making it platform independent.
→ AWT components are heavy weight.	→ Swing components are light weight
→ AWT does not follow MVC (Model View Controller).	→ Swing is based on MVC.
→ AWT components requires java.awt. package.	→ Swing components require javax.swing.package
→ AWT does not have JTable and JTabbed pane.	→ Swing has JTable and JTabbed pane.
→ AWT is platform dependent hence not very different look and feel are supported.	→ Swing can have different look and feel.
→ AWT occupies more memory space.	→ Swing occupies less memory space.
→ AWT is less powerful than Swing.	→ Swing is extension of AWT thus many drawbacks are removed.
→ Look and Feel of AWT is dependent on platform.	→ Look and feel of swing is independent of underlying platform and operating system.
→ AWT components depends on native code (peers) to function properly. That's why these components are "Heavy Weight Components".	→ Swing has no such dependency. Hence these components are "Light Weight Components".
→ Applet can not have menu.	→ JApplet can contain menu.
→ Menu item can not have image or radio button or check boxes.	→ Menu item can have images or radio button or check boxes.
→ Has less advanced components.	→ Has more advanced components
→ Execution is slower.	→ Execution is faster.
→ AWT doesn't follows MVC (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	→ Swing follows MVC.


java.lang
Object
java.awt
Component
Container
Window
Panel
Frame
Dialog
java.applet
Applet
javax.swing
JFrame
JDialog
JWindow
JApplet
A Dialog Box
Enter a number

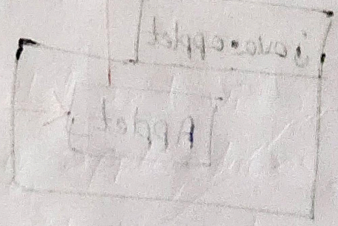

Containers
Components
Frame
(Top-level Container)
Panel
(Partitions)
AWT Test
Label
TextField1
Button1
Button2
Button3
Label
TextField
Button

Limitations of AWT:

The AWT defines a basic set of controls, Windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that is translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by java. Because the AWT components use native code resources, they are referred to as heavy weight.

The use of native peers led to several problems. First, because of variations between ~~platforms~~ operating systems, a component might look, or even act, differently on different platforms. This variability threatened java's philosophy: write once, run anywhere. Second, the look and feel of each component was fixed and could not be changed. Third, the use of heavyweight components caused some frustrating restrictions.

Due to these limitations Swing came and was integrated to java.



Swing is built on the AWT.
Two key Swing features are:

Swing components are light weight,
Swing supports a pluggable look and feel.

→ Platform-dependent: AWT is built on top of the native GUI toolkits of the underlying operating system, which means that the look and feel of AWT components can be different on different platforms.

→ Limited customization: The components in AWT have a limited set of properties that can be modified, making it difficult to customize the look and feel of the components.

→ Limited layout management: AWT's layout management is basic and it can be difficult to create complex layouts with it.

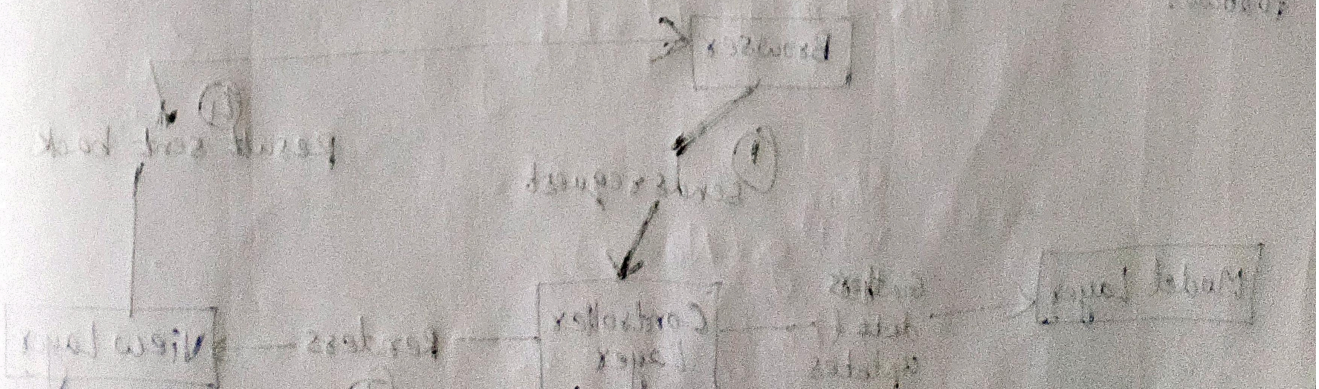
→ Limited functionality: AWT components have a limited set of functionality, and many of the more advanced features found in other toolkits are not available in AWT.

→ Poor performance: AWT components are heavy-weight and can consume a lot of resources, making them less performat than other toolkits, especially when handling large numbers of components.

→ Fewer component types: tables and trees these important components are missing. They are commonly used in desktop applications.
Two very important components trees and tables are not present.

→ Lack of sich component features: The button does not support pictures.

→ No extensibility: The components of AWT are local components. The AWT class instance in the JVM is actually just a reference to the local component. The only extension point is the AWT's canvas component, which creates a custom component from ~~ser~~ scratch. However, it is not possible to inherit and reuse an existing AWT component.
Extensibility is not possible as it is platform dependent.



MVC Architecture:

The Model-View-Controller (MVC) is a well-known design pattern in the web development field. It is way to organize our code. It specifies that a program or application shall consist of data model, presentation information and control information. The MVC pattern needs all these components to be seperated as different objects.

What is MVC architecture in Java?

The model designs based on the MVC architecture follow MVC design pattern. The application logic is separated from the user interface while designing the software using model designs.

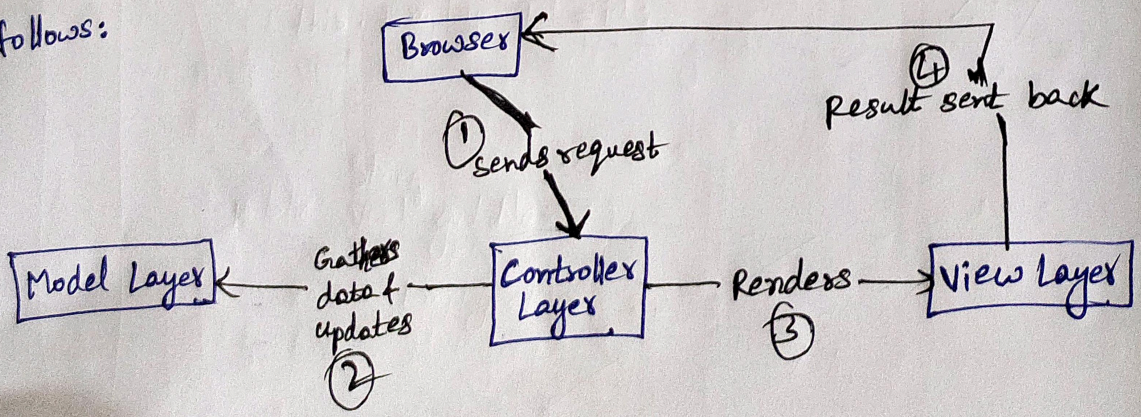
The MVC pattern architecture consists of these three layers:

→ Model: It represents the business layer of applications. It is an object to carry the data that can also contain the logic to update controller if data is changed.

→ View: It represents the presentation layer of application. It is used to visualize the data that the model contains.

→ Controller: It works on both the model and view. It is used to manage the flow of applications, ie, data flow in the model object and to update the view whenever data is changed.

In Java programming, the Model contains the simple Java classes, the View used to display the data and the Controller contains the servlets. Due to this separation the user requests are processed as follows:



1) A client (browser) sends a request to the controller on the server side, for a page.
2) The controller than calls the model. It gathers the requested data.
3) Then the controller transfers the data retrieved to the views layer.
4) Now the result is sent back to the browser (client) by the views.

Advantages of MVC Architecture:

The advantages of MVC architecture are as follows:

→ MVC has the feature of scalability that in turn helps the growth of applications.
→ The components are easy to maintain because there is less dependency.
→ A model can be reused by multiple views that provides reusability of code.
→ The developers can work with the three layers (Model, View, and Controller) simultaneously.
→ Using MVC, the application becomes more understandable.
→ Using MVC, each layer is maintained seperately therefore we do not require to deal with massive code.
→ The extending and testing of application is easier.

Implementation of MVC using Java:

To implement MVC pattern in Java, we are required to create the following three classes.

→ Employee Class, will act as model layer.
→ EmployeeView Class, will act as a view layer.
→ EmployeeController Class, will act a controller layer.

MVC Architecture Layers:

Model Layer:

The Model in the MVC design pattern acts as a data layer for the application. It represents the business logic for application and also the state of application. The model object fetch and store the model state in the database. Using the model layers, rules are applied to the data that represents the concepts of application.

Let's consider the following code snippet that creates a which is also the first step to implement MVC pattern.

Employee.java

```
// class that represents model
public class Employee
{
    //declaring the variables
    private String EmployeeName;
    private String EmployeeId;
    private String EmployeeDepartment;

    // defining getter and setter methods
    public String getId()
    {
        return EmployeeId;
    }

    public void setId(String id)
    {
        this.EmployeeId = id;
    }

    public String getName()
    {
        return EmployeeName;
    }
```

```
    public void setName(String name)
    {
        this.EmployeeName = name;
    }

    public String getDepartment()
    {
        return EmployeeDepartment;
    }

    public void setDepartment(String Department)
    {
        this.EmployeeDepartment = Department;
    }
}
```

The above code simply consists of getter and setter methods to the Employee class.

View Layer:

As the name depicts, view represents the visualization of data received from the model. The view layer consists of output of application or user interface. It sends the requested data to the client, that is fetched from model layer by controller.

Let's take an example where we create a view using the EmployeeView class.

EmployeeView.java

```
//class which represents the view
public class EmployeeView
{
    //method to display the Employee details
    public void printEmployeeDetails(String EmployeeName, String EmployeeId,
                                     String EmployeeDepartment)
    {
        S.o.p("Employee Details:");
        S.o.p("Name:" +EmployeeName);
        S.o.p("EmployeeID:" +EmployeeId);
        S.o.p("Employee Department:" +EmployeeDepartment);
    }
}
```

Controller Layer:

The controller layer gets the user requests from the view layer and processes them, with the necessary validations. It acts as an interface between Model and View. The requests are then sent to model for data processing. Once they are processed, the data is sent back to the controller and then displayed on the view.

Let's consider the following code snippet that creates the controller using the EmployeeController class.

EmployeeController.java

```
//class which represent the controller
public class EmployeeController
{
    //declaring the variables model and view
    private Employee model;
    private EmployeeView view;

    //constructor to intialize
    public EmployeeController(Employee model, EmployeeView view)
    {
        this.model = model;
        this.view = view;
    }

    //getter and setter methods
    public void setEmployeeName(String name)
    {
        model.setName(name);
    }

    public String getEmployeeName()
    {
        return model.getName();
    }

    public void setEmployeeId(String id)
    {
        model.setId(id);
    }

    public String getEmployeeId()
    {
        return model.getId();
    }
```

```
    public void setEmployeeDepartment(String Department)
    {
        model.setDepartment(Department);
    }

    public String getEmployeeDepartment()
    {
        return model.getDepartment();
    }

    //method to update view
    public void updateView()
    {
        view.printEmployeeDetails(model.getName(), model.getId(),
                                  model.getDepartment());
    }
}
```

Main Class Java file:

The following example displays the main file to implement the MVC architecture. Here, we are using the MVCMain class.

MVCMain.java

```
//main class
public class MVCMain
{
    PSVM(String[] args)
    {
        //fetching the employee record based on the employee_id form
                                                    the database
        Employee model = retriveEmployeeFromDatabase();

        //creating a view to write Employee details on console
        EmployeeView view = new EmployeeView();

        EmployeeController controller = new EmployeeController(
                                                    model, view);

        controller.updateView();
```

```
        //updating the model data
        controller.setEmployeeName("Nirmay");
        S.o.p("\n Employee Details after updating:");
        controller.updateView();
    }

    private static Employee retrieveEmployeeFromDatabase()
    {
        Employee Employee = new Employee();
        Employee.setName("Anu");
        Employee.setId("11");
        Employee.setDepartment("Salesforce");
        return Employee;
    }
}
```

The MVCMain class fetches the employee data from the method where we have entered the values. Then it pushes those values in the model. After that, it initializes the view (EmployeeView.java). When view is initialized, the Controller (EmployeeController.java) is invoked and bind it to Employee class and EmployeeView class. At last the updateView() method (method of controller) update the employee details to be printed to the console.

Output:

```
Employee Details:
Name: Anu
Employee ID: 11
Employee Department: Salesforce

Employee Details after updating:
Name: Nirmay
Employee ID: 11
Employee Department: Salesforce
```

Components: (or) Elements:

→ AWT (Abstract Window Toolkit) → contains collection of components (or) Tools is an API to develop GUI (or) window-based application in java.

→ AWT components are platform dependent ie, components are displayed according to the views of OS.

→ java.awt package provides classes for AWT APIs (Application programming Interfaces) such as

- Textfield
- Label
- TextArea
- CheckBox
- Choice
- List
- ⋮

Useful Methods of components class:

add(Component c)

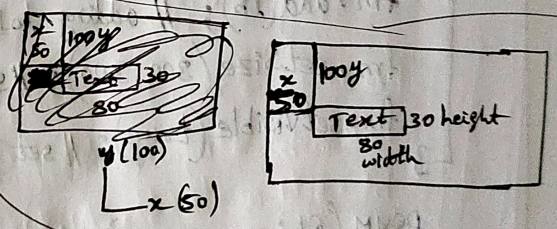
setSize(int width, int height) — 300 × 300

setLayout(LayoutManager M) — location, (Null)
- Flow
- Grid — rows × columns

setVisible(boolean status)
- true (or) false
- true → Frame visible
- false → Frame not visible

setTitle("String Text")

setBounds(x, y, width, height) — 50, 100, 80, 30



Frame:

→ By extending Frame class (inheritance)

Ex: class Example extends Frame

```
{
}
```

→ By creating the object of Frame class (Association)

Ex: class Example

```
{
    Frame obj = new Frame();
    obj.setSize();
}
```

Frame: (container)

Frame is a subclass of Window and have resizing canvas. It is a container that contains several different components like button, title bar, textfield, label etc. In Java, most of the AWT applications are created using Frame window. Frame class has two different constructors,

Frame() throws HeadlessException

Frame(String title) throws HeadlessException

Creating a Frame:

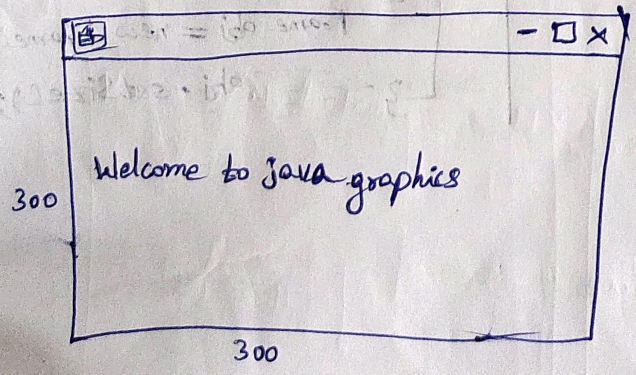
There are two ways to create a Frame. They are

1) By Instantiating Frame class
2) By extending Frame class

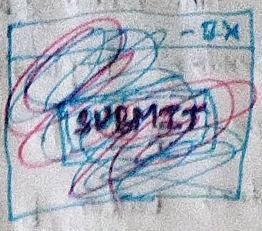
1) Creating Frame Window by Instantiating (→ instance of) Frame class:

```
import java.awt.*;
public class Testawt
{
    Testawt()
    {
        Frame fm = new Frame(); // Creating a frame
        Label lb = new Label("Welcome to java graphics"); // creating a label
        fm.add(lb); // adding label to the frame
        fm.setSize(300,300); // setting frame size
        fm.setVisible(true); // set frame visibility true
    }
    PSVM(String args[])
    {
        Testawt ta = new Testawt();
    }
}
```

Output:

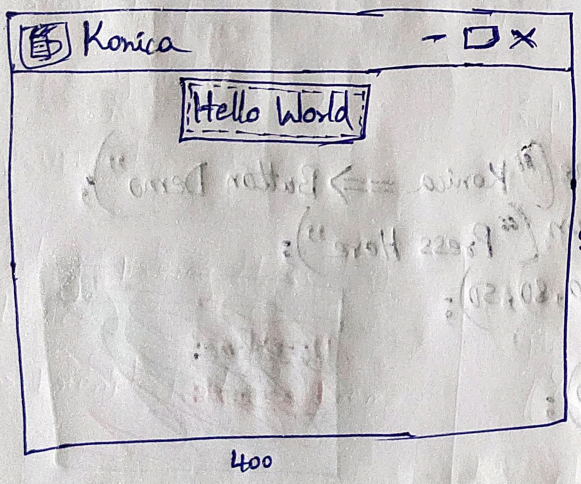


2) Creating Frame Window by extending Frame class:




```
package testawt;
import java.awt.*;
import java.awt.event.*;
public class Testawt extends Frame
{
    public Testawt()
    {
        Button btn = new Button("Hello World");
        add(btn); //adding a new Button
        setSize(400,500); // setting size
        setTitle("Konica"); // setting title
        setLayout(new FlowLayout()); // set default layout for frame
        setVisible(true); // set frame visibility true
    }
    PSVM(String[] args)
    {
        Testawt ta = new Testawt(); // creating a frame
    }
}
```

output:



Points to Remember:

1) While creating a frame (either by instantiating or extending Frame class), Following two attributes are must for visibility of the frame:
   → setSize(int width, int height);
   → setVisible(true);
2) When you create other components like Buttons, TextFields, etc. Then you need to add it to the frame by using the method - add(Component's Object);
3) You can add the following method also for resizing the frame -
   setResizable(true);

Button: (Component)

The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

Syntax: `Button b = new Button("Text");`

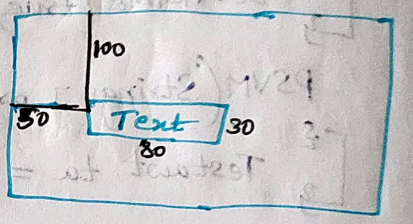
(or)

```
Button b1, b2;
b1 = new Button("Text");
b.setBounds(50, 100, 80, 30);
```

`setBounds(int x, int y, int width, int height)`

This method is used to declare location, width & height of all components of AWT.

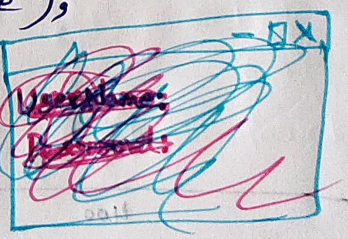
Ex: `setBounds(50, 100, 80, 30);`



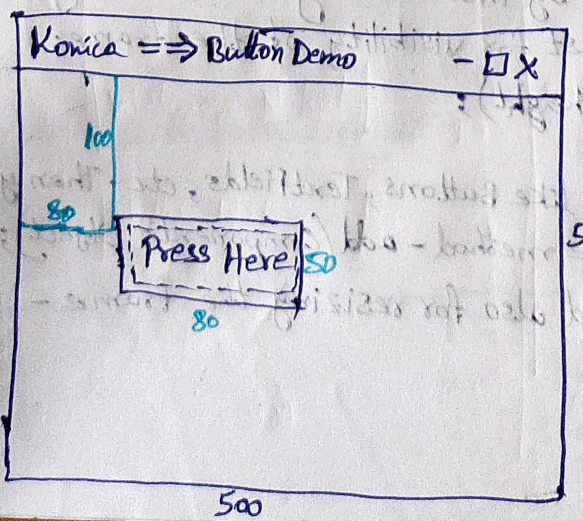
width, height

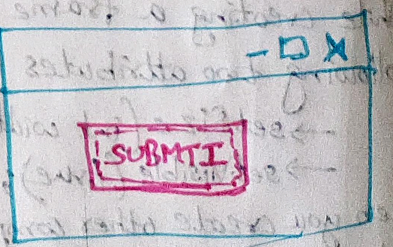
→ Program:

```
import java.awt.*;
public class ButtonDemo1
{
    psvm(String[] args)
    {
        Frame f1 = new Frame("Konica ==> Button Demo");
        Button b1 = new Button("Press Here");
        b1.setBounds(80, 200, 80, 50);
        f1.add(b1);
        f1.setSize(500, 500);
        f1.setLayout(null);
        f1.setVisible(true);
    }
}
```



Output:





Label: (Component)

The Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

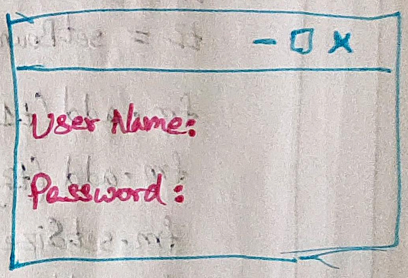
syntax: `Label l1 = new Label("Text");`

(or)

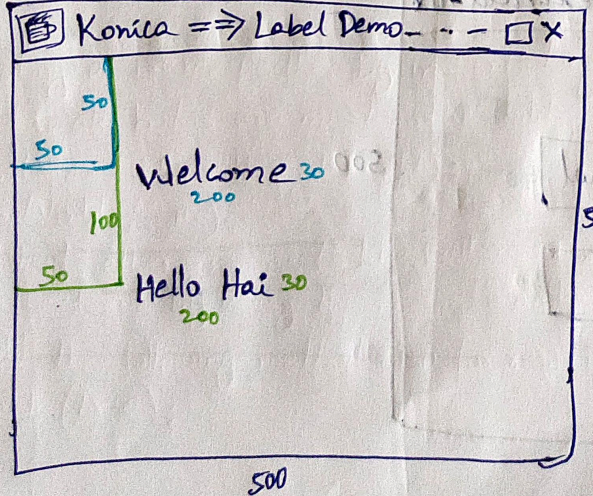
```
Label l1, l2;
l1 = new Label("Text");
```

→ Program:

```
import java.awt.*;
class LabelDemo1
{
    PSVM(String args[])
    {
        Frame fm = new Frame("Konica ==> Label Demo");
        Label l1, l2;
        l1 = new Label("Welcome");
        l1.setBounds(50, 50, 200, 30);
        l2.new Label("Hello Hai");
        l2.setBounds(50, 100, 200, 30);
        fm.add(l1);
        fm.add(l2);
        fm.setSize(500, 500);
        fm.setLayout(null);
        fm.setVisible(true);
    }
}
```



Output:



TextField: (component)

The TextField class is a text component that allows the editing of a single line text. It is used for displaying single line text.

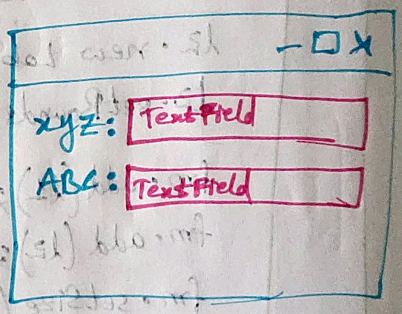
Syntax: `TextField t1 = new TextField("Text");`

(or)

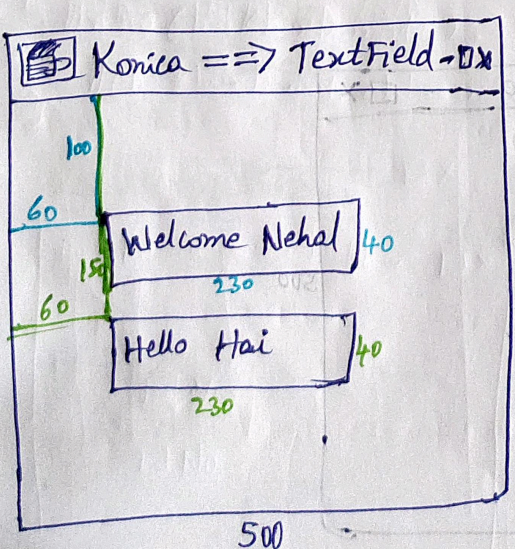
```
TextField t1, t2;
t1 = new TextField("Text");
```

→ **Program:**

```
import java.awt.*;
class TextFieldDemo1
{
  PSVM(String args[])
  {
    Frame fm = new Frame("Konica ==> TextField");
    TextField t1, t2;
    t1 = new TextField("Welcome Nehal");
    t1.setBounds(60, 100, 230, 40);
    t2 = new TextField("Hello Hai");
    t2 = setBounds(60, 150, 230, 40);
    fm.add(t1);
    fm.add(t2);
    fm.setSize(500, 500);
    fm.setLayout(null);
    fm.setVisible(true);
  }
}
```



output:



TextArea: (Component)

The TextArea class is a multi-line region that displays text. It allows the editing of multiple line text. It is used for displaying multiple-line text.

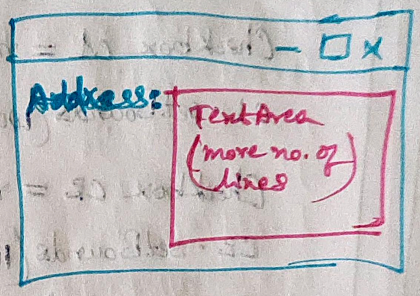
Syntax: `TextArea t1 = new TextArea("Text");`

(or)

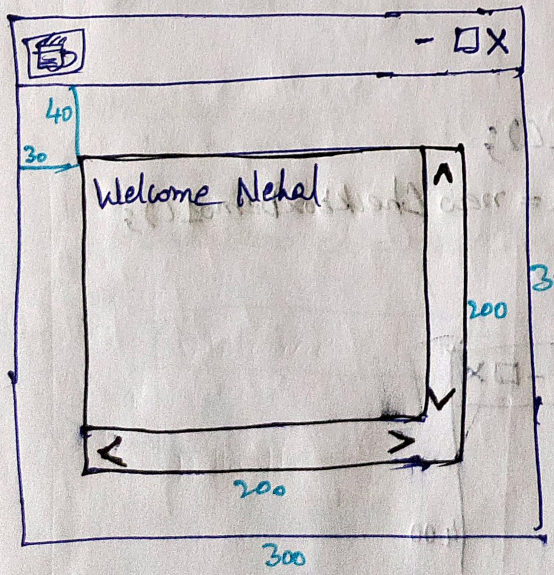
```
TextArea t1, t2;
t1 = new TextArea("Text");
```

→ Program:

```
import java.awt.*;
public class TextAreaDemo1
{
    TextAreaDemo1()
    {
        Frame fm = new Frame();
        TextArea ta = new TextArea("Welcome Nehal");
        ta.setBounds(30,40,200,200);
        fm.add(ta);
        fm.setSize(300,300);
        fm.setLayout(null);
        fm.setVisible(true);
    }
    PSVM(String args[])
    {
        new TextAreaDemo1();
        // TextAreaDemo1 tad = new TextAreaDemo1();
    }
}
```



output:



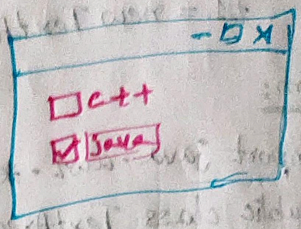
Checkbox: (Component)

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off to "on".

Syntax: `Checkbox c1 = new Checkbox("Text");`

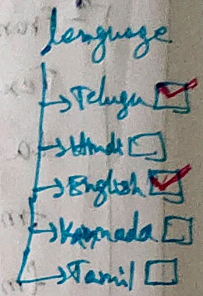
(or)

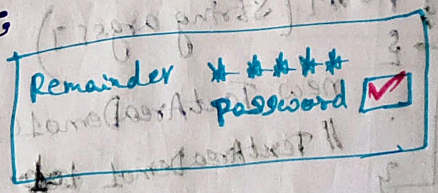
```
Checkbox c1, c2;
c1 = new Checkbox("Text");
```



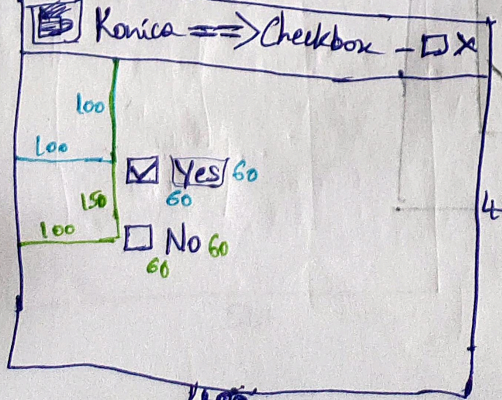
→ **Program:**

```
import java.awt.*;
public class CheckboxDemo1
{
    CheckboxDemo1()
    {
        Frame fm = new Fram("Konica ==> CheckBox");
        Checkbox c1 = new Checkbox("Yes", true);
        c1.setBounds(100, 100, 60, 60);
        Checkbox c2 = new Checkbox("No");
        c2.setBounds(100, 150, 60, 60);
        fm.add(c1);
        fm.add(c2);
        fm.setSize(400, 400);
        fm.setLayout(null);
        fm.setVisible(true);
    }
    PSVM(String args[])
    {
        new CheckboxDemo1();
        // CheckboxDemo1 cbd = new CheckboxDemo1();
    }
}
```





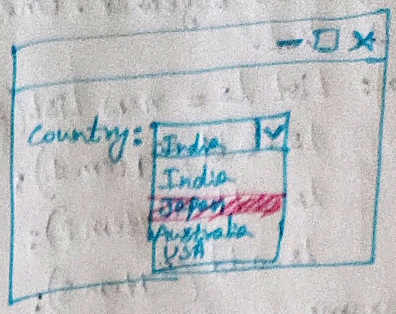
output:



Choice: (component)

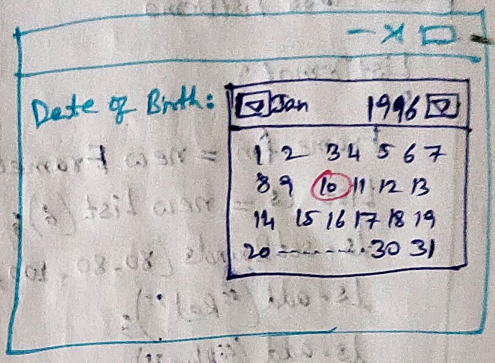
The Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu.

Syntax:
```
Choice c = new Choice();
c.add("Item1");
c.add("Item2");
c.add("Item3");
```

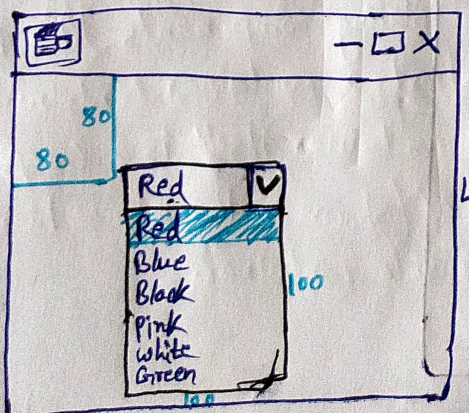


→ Program:

```
import java.awt.*;
public class ChoiceDemo
{
    ChoiceDemo()
    {
        Frame fm = new Frame();
        Choice c = new Choice();
        c.setBounds(80,80,100,100);
        c.add("Red");
        c.add("Blue");
        c.add("Black");
        c.add("Pink");
        c.add("White");
        c.add("Green");
        fm.add(c);
        fm.setSize(400,400);
        fm.setLayout(null);
        fm.setVisible(true);
    }
    PSVM(String args[])
    {
        new ChoiceDemo();
        // ChoiceDemo cd = new ChoiceDemo();
    }
}
```



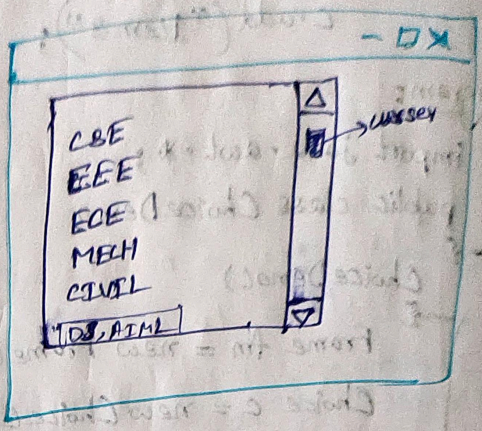
output:



List: (Component)

The List class represents a list of text items. By the help of list, user can choose either one item or multiple items. It is used to represent a list of items together. One or more than one item can be selected from the list.

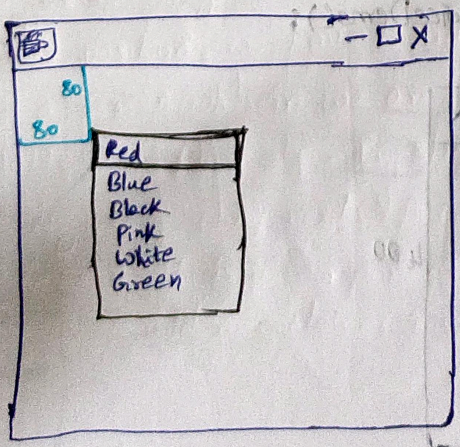
Syntax:
```
List ls = new List(Size);
ls.add("Item 1");
ls.add("Item 2");
ls.add("Item 3");
```



→ Program:

```
import java.awt.*;
public class ListDemo
{
    ListDemo();
    {
        Frame fm = new Frame();
        List ls = new List(6);
        ls.setBounds(80,80,100,100);
        ls.add("Red");
        ls.add("Blue");
        ls.add("Black");
        ls.add("Pink");
        ls.add("White");
        ls.add("Green");
        fm.add(ls);
        fm.setSize(400,400);
        fm.setLayout(null);
        fm.setVisible(true);
    }
    psvm(String args[])
    {
        new ListDemo();
        // ListDemo ld = new ListDemo();
    }
}
```

Output:



Closing Window:

→ AWT does not allow the user to close windows directly.

→ we need code to close window.

Method-I

```
//code for Close windows
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent we)
{
    System.exit(0);
}
});
```

Note: We need to import one package "java.awt.event.*"

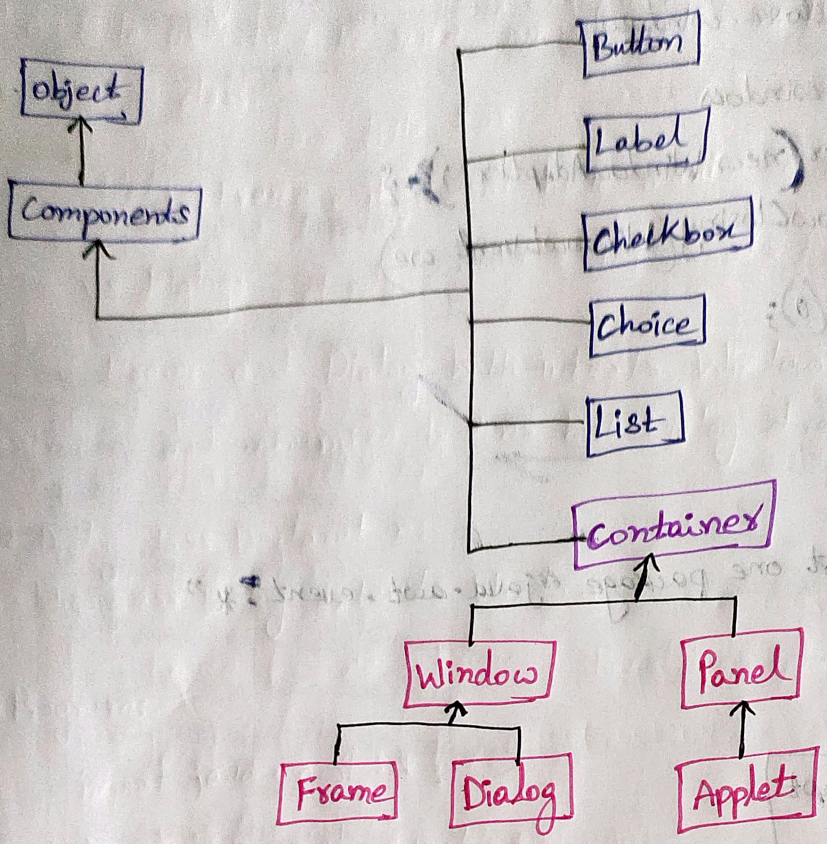
Method-II

Command prompt
cntrl c

To close window.
If we don't want to write the code then simple press cntrl+c.

Containers: (Component)

Java AWT Hierarchy:



→ The container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as Container such as Frame, Dialog and Panel.

→ It is basically a screen where the components are placed at their specific locations. Thus it contains and controls the layout of components.

→ Note: A container itself is a component (see the above diagram), therefore we can add a container inside container.

→ All the elements like buttons, text fields, scrollbars etc. are known as components. In AWT we have classes for each components as shown in the above diagram. To have everything placed on a screen to a particular position, we have to add them to a container. A Container is like a screen wherein we are placing components like buttons, textfields, checkbox etc. In short a container contains and controls the layout of components. ~~A container itself is a components.~~ A Container itself is a component (shown in the above hierarchy diagram) thus we can add a container inside container.

Types of Containers:

A Container is a place wherein we add components like text field, button, checkbox etc. There are four types of containers available in AWT: Window, Frame, Dialog and Panel. As shown in the hierarchy diagram above, Frame and Dialog are subclasses of Window class.

→ **Window:** An instance of the Window class has no border and no title. The Window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

→ **Frame:** A frame has title, border and menu bars. It can contain several components like buttons, text fields, scrollbars etc. This is most widely used container while developing an application in AWT.

→ **Dialog:** Dialog class has border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.

→ **Panel:** Panel does not contain title bar, menu bar or border. It is a generic container for holding components. An instance of the Panel class provides a container to which to add components. It can have other components like button, text field etc.

Dialog: (Container)

The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.

Unlike Frame, it doesn't have maximize and minimize buttons.

Frame vs Dialog:

Frame and Dialog both inherits Window class. Frame has maximize and minimize buttons but Dialog doesn't have.

AWT Dialog class Declaration:

public class Dialog extends Window

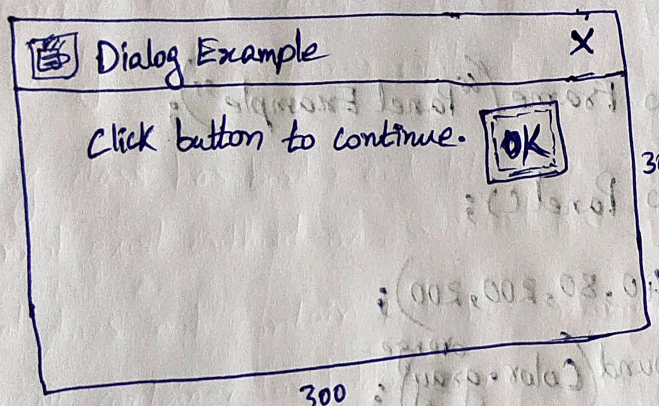
→ Program:

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample
{
    private static Dialog d;
    DialogExample()
    {
        Frame f = new Frame();
        d = new Dialog (f, "Dialog Example", true);
        d.setLayout (new FlowLayout());
        Button b = new Button("OK");
        b.addActionListener(new ActionListener()
        {
            public void actionPerformed (ActionEvents e)
            {
                DialogExample.d.setVisible(false);
            }
        });
```

```
    d.add(new Label("Click button to continue."));
    d.add(b);
    d.setSize(300,300);
    d.setVisible(true);
  }

  public
  PSVM(String args[])
  {
    new DialogExample();
  }
}
```

output:



Panel: (Container)

The Panel is a simplest container class. It provides space in which an application can attach any other component. It inherits the Container class.

It doesn't have title bar.

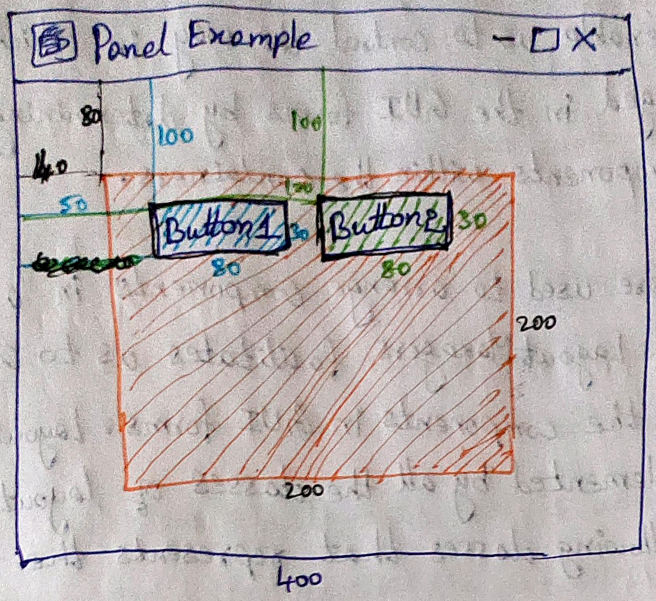
AWT Panel class declaration:

public class Panel extends Container implements Accessible

→ **Program:**

```
import java.awt.*;
public class PanelExample
{
    PanelExample()
    {
        Frame f = new Frame("Panel Example");
        Panel p = new Panel();
        p.setBounds(40,80,200,200);
        p.setBackground(Color.orange);
        Button b1 = new Button("Button 1");
        b1.setBounds(50,100,80,30);
        b1.setBackground(Color.blue);
        Button b2 = new Button("Button 2");
        b2.setBounds(100,100,80,30);
        b2.setBackground(Color.green);
        p.add(b1);
        p.add(b2);
        f.add(p);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
    PSVM(String args[])
    {
        new PanelExample();
    }
}
```

output:


Panel Example
80
100
100
120
50
Button1
30
Button2
30
80
80
200
400
200
400

Understanding Layout Managers:

The Layout managers enable us to control the way in which visual components are arranged in the GUI forms by determining the size and position of components within the containers.

The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represents the layout managers:

1) java.awt.FlowLayout
2) java.awt.BorderLayout
3) java.awt.GridLayout
4) java.awt.CardLayout
5) java.awt.GridBagLayout
6) javax.swing.BoxLayout
7) javax.swing.GroupLayout
8) javax.swing.ScrollPanelLayout
9) javax.swing.SpringLayout etc.

FlowLayout:

FlowLayout arranges the components in a container like the words on a page. It fills the top line from left to right and top to bottom. The components are arranged in the order as they are added ie, first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be left, center or right aligned.

The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

Fields of FlowLayout class:

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 of FlowLayout class:

1) FlowLayout(): creates a flow layout with center alignment and a default 5 unit horizontal and vertical gap.
2) FlowLayout(int align): creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3) FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example of FlowLayout class: Using FlowLayout() constructor.

FileName: FlowLayoutExample.java

→ Program:

```
//import statements
import java.awt.*;
import javax.swing.*;
public class FlowLayoutExample
{
    JFrame frameObj;
    //constructor
    FlowLayoutExample()
    {
        //creating a frame object
        frameObj = new JFrame();

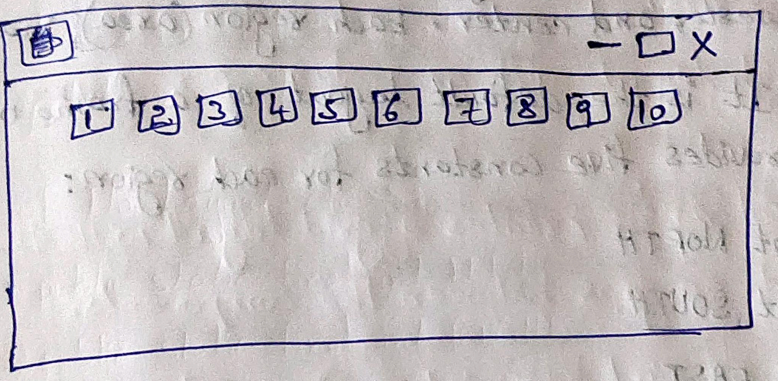
        //creating the buttons
        JButton b1 = new JButton("1");
        JButton b2 = new JButton("2");
        JButton b3 = new JButton("3");
        JButton b4 = new JButton("4");
        JButton b5 = new JButton("5");
        JButton b6 = new JButton("6");
        JButton b7 = new JButton("7");
        JButton b8 = new JButton("8");
        JButton b9 = new JButton("9");
        JButton b10 = new JButton("10");

        //adding the buttons to frame
        frameObj.add(b1); frameObj.add(b2);
        frameObj.add(b3); frameObj.add(b4);
        frameObj.add(b5); frameObj.add(b6);
        frameObj.add(b7); frameObj.add(b8);
        frameObj.add(b9); frameObj.add(b10);

        //parameter less constructor is used
        // therefore, alignment is center
        // horizontal as well as the vertical gap is 5 units.
        frameObj.setLayout(new FlowLayout());
```

```
    frameObj.setSize(300,300);
    frameObj.setVisible(true);
  }
  main method
  PSVM(String args[])
  {
    new FlowLayoutExample();
  }
}
```

Output:



Border Layout:

BorderLayout arranges all the components along the edges or the middle of the container ie., top, bottom, right and left edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the center gets neither its preferred height or width. It covers the remaining area of the container.

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

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 of BorderLayout class:

1) BorderLayout(): creates a border layout but with no gaps between the components.

2) BorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class: Using BorderLayout() constructor:

File Name: Border.java

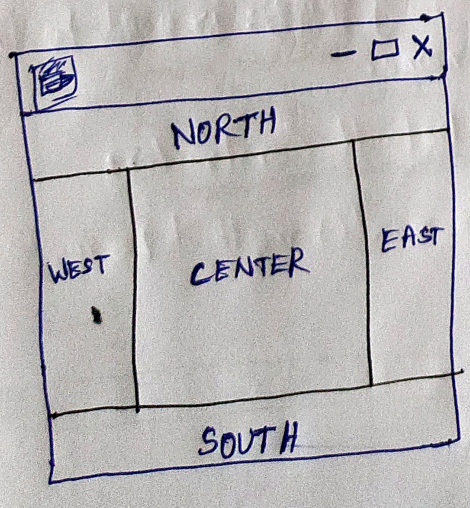
→ Program:

```
import java.awt.*;
import javax.swing.*;
public class Border
{
    JFrame f;
    Border()
    {
        f = new JFrame();
        //creating buttons
        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");

        // Buttons placed in the particular Direction
        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);
    }
    PSVM(String args[])
    {
        new Border();
    }
}
```

Output:



Grid Layout:

GridLayout arranges all the components in a grid of equally sized cells, adding them from left to right and top to bottom. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.

The Java GridLayout class is used to arrange the components in a rectangular grid. One component is displayed in each rectangle.

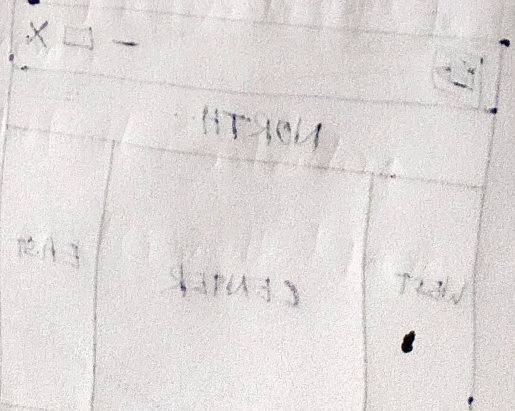
Constructors of GridLayout class:

1) GridLayout(): creates a grid layout with one column per component in a row.
2) GridLayout(int rows, int columns): creates a grid layout with the given rows and columns but no gaps between the components.
3) 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 of GridLayout class: Using GridLayout() Constructor:

The GridLayout() constructor creates only one row. The following example shows the usage of the parameterless constructor.

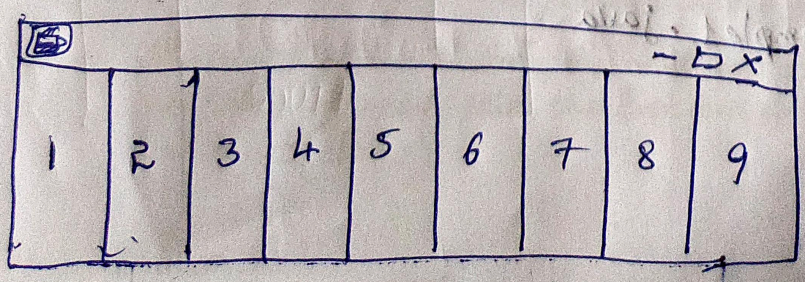
File Name: GridLayoutExample.java



→ Program:

```
//import statements
import java.awt.*;
import javax.swing.*;
public class GridLayoutExample
{
    JFrame frameObj;
    //constructor
    GridLayoutExample()
    {
        frameObj = new JFrame();
        //creating 9 buttons
        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");
        //adding buttons to the frame
        //since, we are using the parameterless constructor, therefore;
        //the number of columns is equal to the number of buttons we
        //are adding to the frame. The row count remains one.
        frameObj.add(b1); frameObj.add(b2); frameObj.add(b3);
        frameObj.add(b4); frameObj.add(b5); frameObj.add(b6);
        frameObj.add(b7); frameObj.add(b8); frameObj.add(b9);
        //setting the grid layout using the parameterless constructor
        frameObj.setLayout(new GridLayout());
        frameObj.setSize(300,300);
        frameObj.setVisible(true);
    }
    //main method
    PSVM(String args[])
    {
        new GridLayoutExample();
    }
}
```

Output:



Card Layout:

CardLayout arranges two or more components having the same size. The components are arranged in a deck, where all the cards of the same size and the only top card are visible at any time. The first component added in the container will be kept at the top of the deck. The default gap at the left, right, top and bottom edges are zero and the card components are displayed either horizontally or vertically.

The Java CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout Class:

1) CardLayout(): creates a card layout with zero horizontal and vertical gap.
2) CardLayout(int hgap, int vgap): creates a card layout with the given horizontal and vertical gap.

Commonly Used Methods of CardLayout Class:

→ public void next(Container parent): is used to flip to the next card of the given container.
→ public void previous(Container parent): is used to flip to the previous card of the given container.
→ public void first(Container parent): is used to flip to the first card of the given container.
→ public void last(Container parent): is used to filp to the last card of the given container.
→ public void show(Container parent, String name): is used to flip to the specified card with the given name.

Example of CardLayout Class: Using Default Constructor:

The following program uses the next() method to move to the next card of the container.

FileName: CardLayoutExample1.java

→ Program:

```
//import statements
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class CardLayoutExample1 exends JFrame implements ActionListener
{
    CardLayout crd;

    //button variables to hold the references of buttons
    JButton b1, b2, b3;
    Container cPane;

    //Constructor of the class
    CardLayoutExample1()
    {
        cPane = getContentPane();
        //default constructor used
        //therefore, components will
        //cover the whole area
        crd = new CardLayout();
        cPane.setLayout(crd);

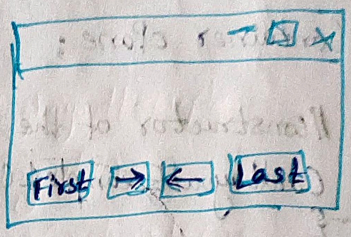
        //creating the buttons
        b1 = new JButton("Apple");
        b2 = new JButton("Boy");
        b3 = new JButton("Cat");

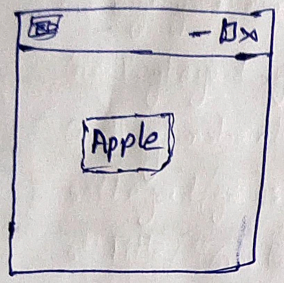
        //adding listeners to it
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);

        cPane.add("a", b1); // first card is the button b1
        cPane.add("b", b2); // first card is the button b2
        cPane.add("c", b3); // first card is the button b3
    }
    public void actionPerformed(ActionEvent e)
    {
        //Upon clicking the button, the next card of the container is shown
        //after the last card, again, the first card of the container is shown upon clicking
        crd.next(cPane);
    }
```

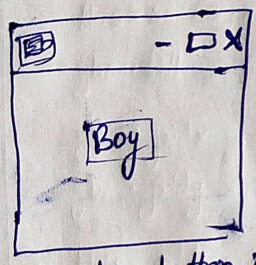
```
//main method
PSVM(String args[])
{
    //creating an object of the class CardLayoutExample1
    CardLayoutExample1 crdl = new CardLayoutExample1();
    //size is 300 * 300
    crdl.setSize(300,300);
    crdl.setVisible(true);
    crdl.setDefaultCloseOperation(EXIT_ON_CLOSE);
}
```

Output:

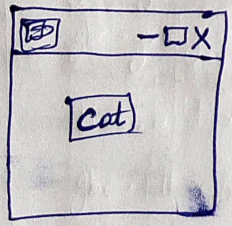




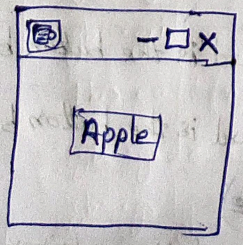
when the button named apple is clicked, we get.



When the boy button is clicked, we get



Again, we reach the first card of the container if the cat button is clicked, and the cycle continues.



GridBagLayout:

It is a powerful layout which arranges all the components in a grid of cells and maintains the aspect of the object whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.

The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline

The components may not be of the same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells. Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints. With the help of the constraints object, we arrange the component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine the component's size. GridBagLayout components are also arranged in the rectangular grid but can have many different sizes and can occupy multiple rows or columns.

Constructor:

→ GridBagLayout(): The parameterless constructor is used to create a grid bag layout manager.

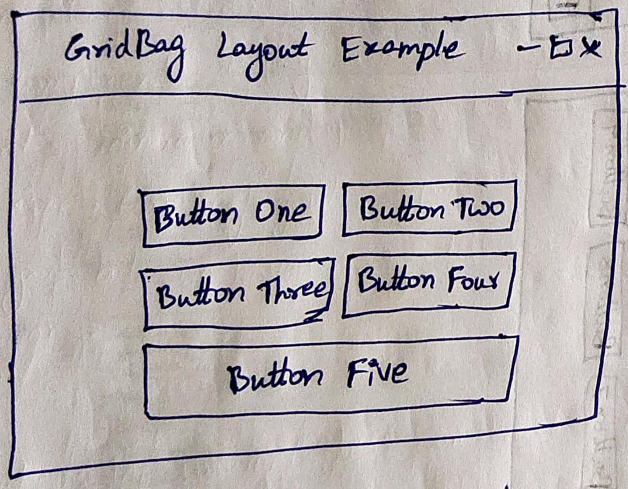
→ Program:

FileName: GridBagLayoutExample.java

```
import java.awt.Button;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.*;
public class GridBagLayoutExample extends JFrame
{
    PSVM(String args[])
    {
        GridBagLayoutExample a = new GridBagLayoutExample();
    }
    public GridBagLayoutExample()
    {
        GridBagLayoutExample = new GridBagLayoutExample
        GridBagLayout grid = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(grid);
        setTitle("GridBag Layout Example");
        GridBagLayout layout = new GridBagLayout();
        this.setLayout(layout);
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.gridx = 0;
        gbc.gridy = 0;
        this.add(new Button("Button One"), gbc);
        gbc.gridx = 1;
        gbc.gridy = 0;
        this.add(new Button("Button Two"), gbc);
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.ipady = 20;
        gbc.gridx = 0;
        gbc.gridy = 1;
        this.add(new Button("Button Three"), gbc);
        gbc.gridx = 1;
        gbc.gridy = 1;
        this.add(new Button("Button Four"), gbc);
```

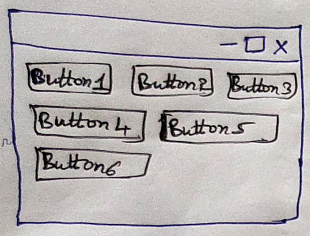
```
gbc.gridx = 0;
gbc.gridy = 2;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridwidth = 2;
this.add(new Button("Button Five"), gbc);
setSize(300,300);
setPreferredSize(getSize());
setVisible(true);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
}
```

output:

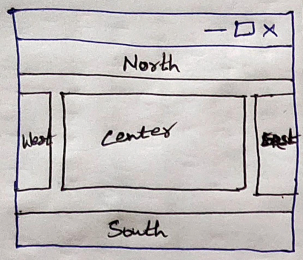


Layout Manager

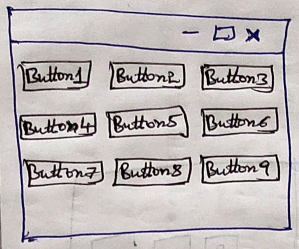
Flow Layout



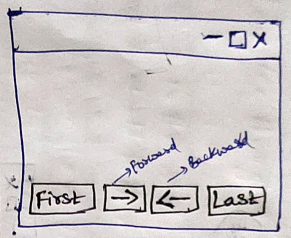
Border Layout

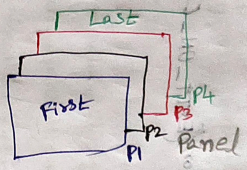


Grid Layout

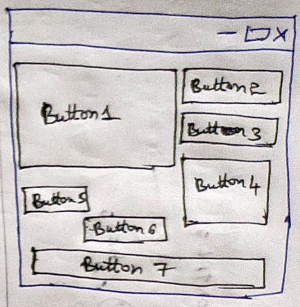


Card Layout

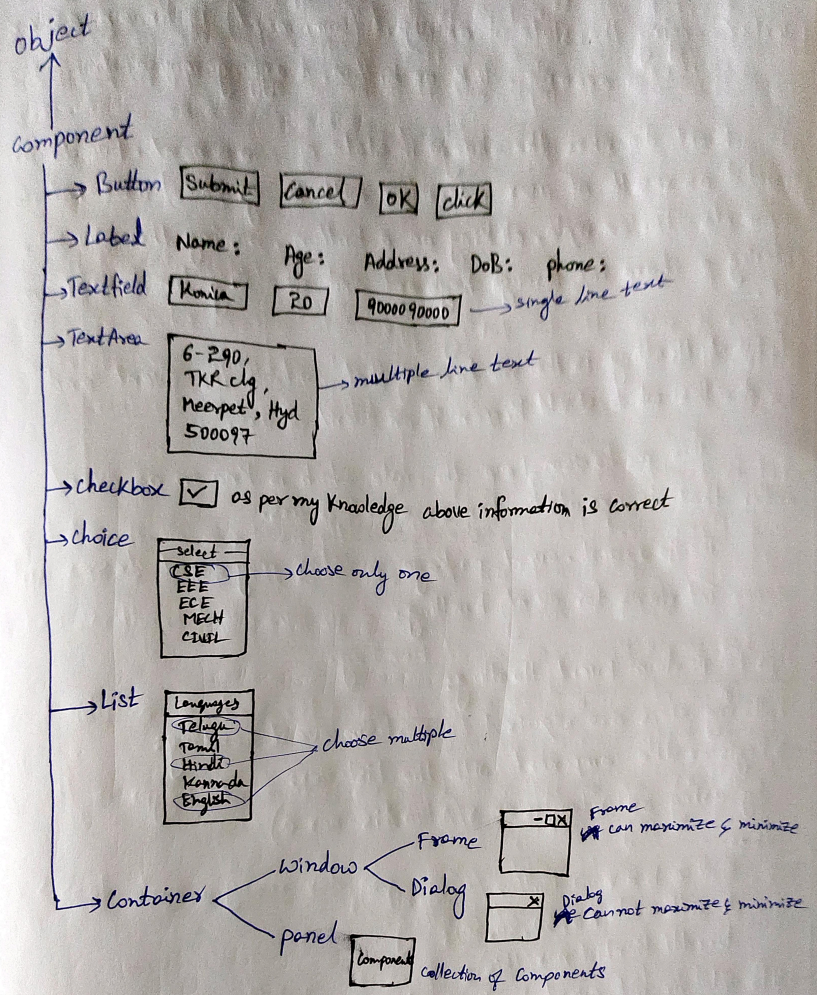


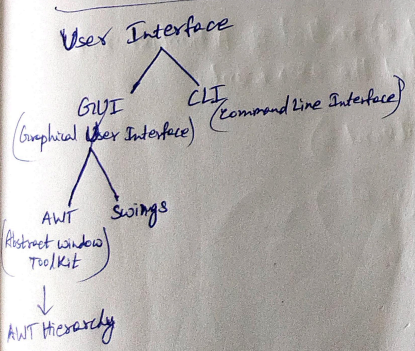


GridBag Layout



AWT Hierarchy:





Event Handling:

Event: Changing the state of an object (component) is known as an event. For example, click on button, dragging mouse etc.

Event describes the change in state of component. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard and selecting an item from list.

Def: Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.

This mechanism have the code which is known as event handler that is executed when an event occurs.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

Steps to perform Event Handling:

Following steps are required to perform event handling:

→ Register the component with the Listener:

By using addActionListener(ActionListener a)

Ex:

```
Button b = new Button("Submit");
b.setBounds(100, 50, 80, 30);
b.addActionListener(this);
```

→ Provide or put event handling code:

By using actionPerformed(ActionEvent e) method of ActionListener interface, we can perform action what the user want.

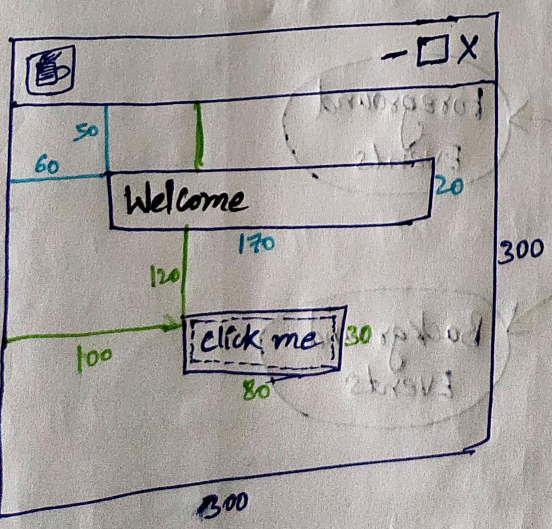
Ex:

```
public void actionPerformed(ActionEvent e)
{
    tf.setText("welcome");
}
```

→ Program:

```
import java.awt.*;
import java.awt.event.*;
class EventExample extends Frame implements ActionListener
{
    Textfield tf;
    EventExample()
    {
        tf = new Textfield(); // create components
        tf.setBounds(60,50,170,20);
        Button b = new Button("click me");
        b.setBounds(100,120,80,30);
        b.addActionListener(this); // register listener & passing current instance
        add(b); // add components and set size, layout and visibility
        add(tf);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        tf.setText("Welcome");
    }
    PSVM(String args[])
    {
        new EventExample();
    }
}
```

output:



Overview:

Event handling in Java is the procedure that controls an event and performs appropriate action if it occurs. The code or set of instructions used to implement it is known as the Event handler.

It consist of two major components:

→ The event source

→ The event listener

The source is the object where the event occurs, and the event listener is responsible for taking appropriate actions when an event occurs. These Listeners must be registered with the source object in order for the listener to receive event notifications.

What is Event Handling?

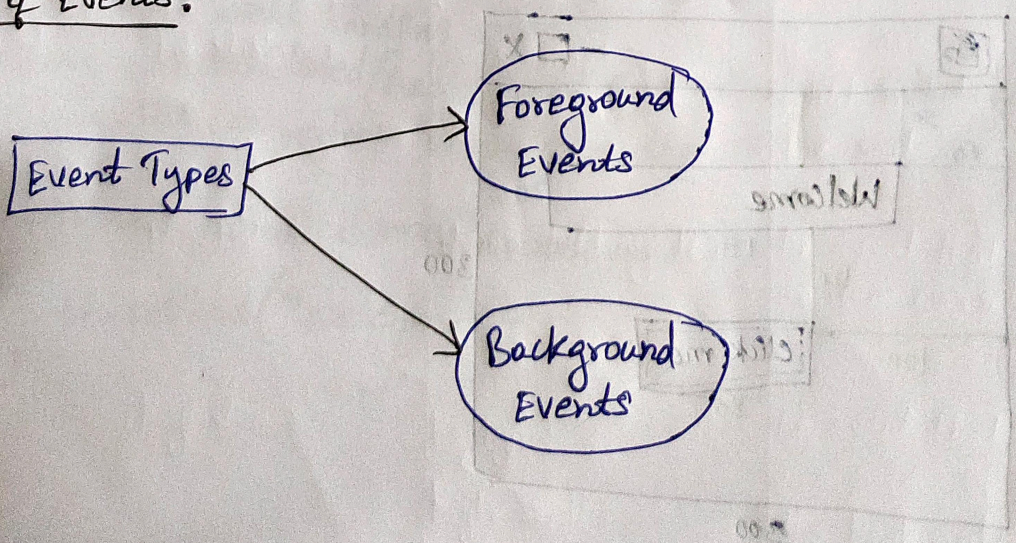
Many event listeners are frequently used; recall the click of a button that takes you to another website or the mouse scroll? All of this is accomplished through the use of various event handlers, and the mechanism is known as event handling.

An event can be defined as changing the state of an object or behaviour by performing actions.

Actions can be button click, cursor movement, keypress through keyboard or page scrolling, etc.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

Types of Events:



→ Foreground Events: These events necessitate the user's direct participation. They are produced as a result of a user interacting with graphical components in a Graphical User Interface. Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor movements etc.

→ Background Events: Background events are those that require end-user interaction. Operating system interrupts and hardware or software failures are examples of background events. Events that don't require interactions of users to generate are known as background events.

The Delegation Event Model: (or) How are Events Handled?

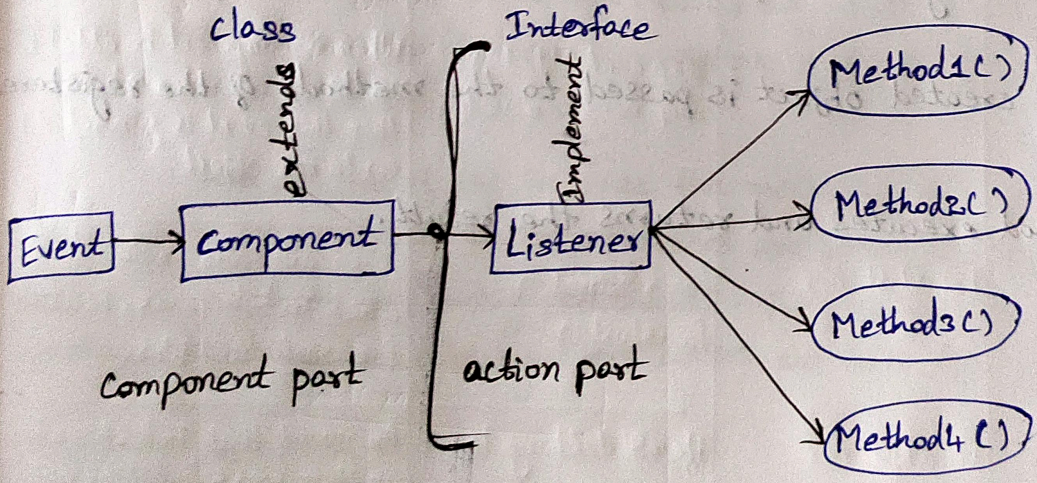
→ Defines standard & consistent mechanism to generate and process events.

→ A source code generates an event and sends it to one or more listeners.

→ The listener simply waits until it receives an event.

→ Once an event is received, the listener processes the event and then returns.

→ In this model, the listeners must register with a source in order to receive an event notification.

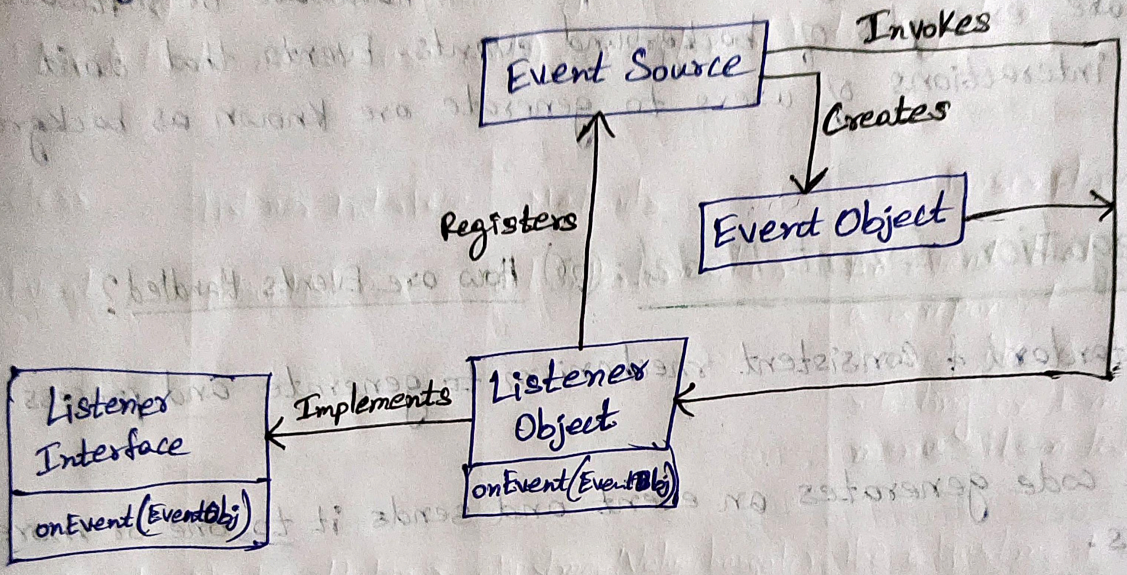


→ Source: Events are generated from the source. These are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.

→ Listeners: Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

To perform Event Handling, we need to register the Source with the listener.

The modern approach to event processing is based on the Delegation Model. It defines a standardized and compatible mechanism for generating and processing events. In this approach, a source generates an event and sends it to one or more listeners. The listener sits and waits for an event to occur. When it gets an event, it is processed by the listener and returned. The user interface elements can delegate the processing of an event to a different function.



Flow of Event Handling:

1) User Interaction with a component is required to generate an event.
2) The object of the respective event class is created automatically after event generation, and it holds all information of the event source.
3) The newly created object is passed to the methods of the registered listener.
4) The method executes and returns the result.

Components of Event Handling:

An Event Model is fundamentally composed of the three elements listed below:

1) → Events Handler (or) Events
2) → Event Sources
3) → Event Listeners

1) Events Handlers (or) Events:

An event handler is a function or method that executes program statements in response to an event. A software program that processes activities such as keystrokes and mouse movements is what an event handler is. Event handlers in Web sites make Web content dynamic.

2) Event Sources:

An object on which an event occurs in referred to as a source. The source is in charge of informing the handler about the event that occurred. There are various sources like buttons, checkboxes, lists, menu-item, choices, scrollbars, text components, windows, etc.

3) Event Listeners:

When an event occurs, an object named an event listener is called. The listeners need two things: first, they must be registered with a source; however, they can be registered with multiple sources to receive event notifications.

Second, it must put in place the methods for receiving and processing notifications. A set of interfaces and processing notifications, defines the methods for dealing with events. The Java.awt.event package contains the following event classes and interfaces.

Event classes:

Registering the source with Listener:

Different classes provide different registration methods.

Syntax: addTypeListener()

where Type represents the type of event.

Ex 1: For KeyEvent we use addKeyListener() to register.

Ex 2: For ActionEvent we use addActionListener() to register.

Event Class	Listener Interface	Methods	Description
ActionEvent	ActionListener	actionPerformed()	An event that indicates that a component-defined action occured like a button click or selecting an item from the menu-item list.
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()	The adjustment event is emitted by an Adjustable object like scrollbar.
ComponentEvent	ComponentListener	componentResized() componentShown() componentMoved() componentHidden()	An event that indicates that a component moved, the size changed or changed its visibility.
ContainerEvent	ContainerListener	componentAdded() componentRemoved()	When a component is added to a container (or) removed from it, then this event is generated by a container object.
FocusEvent	FocusListener	focusGained() focusLost()	These are focus-related events, which include focus, focusin, focusout, and blur.
ItemEvent	ItemListener	itemStateChanged()	An event that indicates whether an item was selected or not.
KeyEvent	KeyListener	keyTyped() KeyPressed() KeyReleased()	An event that occurs due to a sequence of keypresses on the keyboard.

MouseEvent	MouseListener & MouseMotionListener	mousePressed() mouseClicked() mouseEntered() mouseExited() mouseReleased() mouseMoved() mouseDragged()	The events that occur due to the user interaction with the mouse (Pointing Device).
MouseWheelEvent	MouseWheelListener	mouseWheelMoved()	An event that specifies that the mouse wheel was rotated in a component.
TextEvent	TextListener	textChanged()	An event that occurs when an object's text changes.
WindowEvent	WindowListener	windowActivated() windowDeactivated() windowOpened() windowClosed() windowClosing() windowIconified() windowDeiconified()	An event which indicates whether a window has changed its status or not.

Note: As Interfaces contains abstract methods which need to implemented by the registered class to handle events.