Class Loader subsystem with JVM Architectute :-

The three main component of JVM

1) class loader sub system

2) Runtime Data Areas

3) Execution engine

In order to load the required .class file, JVM makes a request to class loader sub system. The class loader sub system follows delegation hierarchy algorithm to load the required .class file from different area.

To load the required .class file we have 3 different kinds of class loader

1) Bootstrap class loader

2) Extension/Platform class loader

3) Application/System class loader

Bootstrap class Loader :-

It is responsible to load the required .class file from java API that means all the predfined

classes .class file will be loaded by Bootstrap class loader.

It the super class of Extension class loader as well as It has the highest priority among all the class loader.

Extension/Platform class Loader :-

It is responsible to load the required .class files from ext (extension) folder. Inside the extension folder we have jar file(Java level zip file) given by some third party or user defined jar file.

It is the super class of Application class loader as well as It has more priority than Application class loader.

Application class Loader :-

It is responsible to load the required .class file from class path level i.e Environment variable. It has lowest priority as well as It is the sub class of Extension class loader.

How Delegation Hierarchy algorithm works :-

Whenever JVM makes a request to class loader sub system to load the required .class file into JVM memory, first of all class loader sub system makes a request to Application class loader , Application class loader will delegate the request to the Extension class loader, Extension class loader will also delegate the request to Bootstrap class loader.

Bootstrap class loader will load the .class file from lib folder(rt.jar) and then by pass the request to extension class loader, Extension class loader will load the .class file from ext folder(*.jar) and by pass the request to Application class loader, It will load the .class file from environment variable into JVM memory.

Note :-

If all the class loaders are unable to load the .class file into JVM memory then we will get a Runtime exception i.e java.lang.ClassNotFoundException

```
class Test
{
  static int x = 100;
 }
```

Linking :-

---

verify :-

It ensures the correctness the .class files, If any suspicious activity is there in the .class file then It will stop the execution immediately by throwing an exception i.e java.lang.VerifyError.

There is something called ByteCodeVerifier, responsible to verify the .class file i.e byte code. Due to this verify module JAVA is highly secure language.

Prepare :-

It will allocate the memory for all the static data member, here all the static data member will get the default values so if we have static int x = 100;

then for x we will get the default value i.e 0.

Resolve :-

All the symbolic references will be converted to direct references.

Initialization :-

In Initialization, all the static data member will get their actual value as well as if any static block is available in the class then the static block will be exceuted here.

static block :-

It is a very special block in java which will be executed at the time of loading the .class file into JVM memory by class loader subsystem.

The main purpose of static block to initialize the static data member of the class.

static block will be executed only once because class loading is possible only once in java.

If we have multiple static blocks in a class then It will be executed according to order.

```
class Test
{
  Test()
        {
           System.out.println("defualt constructor...");
```

```
    }

  {
        System.out.println("instance block...");
  }

  static
  {
        System.out.println("Static block...");
  }
}
class  StaticBlock
{
      public static void main(String[] args)
      {
            Test t1 = new Test();

      }
}
--------------------------
class Demo
{
       static int x;
```

```
	static
	{
		x = 100;
	}

	static
	{
		x = 200;
	}

	static
	{
	System.out.println("x value is :"+x);
	}
}
class StaticBlockOrder
{
	public static void main(String[] args)
	{
		Demo d = new Demo();
	}
}
```

Note :- We can't execute a java program without main method, Upto jdk 1.6 it was possible to execute a java program without main method be writing the static block. From JDK 1.7 onwards

now we can't execute java program without main method.

How to load the .class file Dynamically Or Explicitly :

Java software people has provided a predefined class called Class available in java.lang package.

This class contains a predefined static method forName(), through which we can load the required .class file into JVM memory dynamically.

It throws a checked exception i.e java.lang.ClassNotFoundException

Class.forName("Test"); -> try-catch OR throws

```
class Foo
{
	static
	{
		System.out.println("It is a static block");
	}
}
class DynamicLoading
{
	public static void main(String[] args) throws Exception
```

```
	{
		Class.forName("Foo");
	}
}
```

What is the difference between java.lang.ClassNotFoundException and

---

java.lang.NoClassDefFoundError:-

---

java.lang.ClassNotFoundException :-

It occurs when we try to load the required .class file at runtime by using Class.forName() statement and if the required .class file is not available at runtime then we will get an exception i.e java.lang.ClassNotFoundException

```
class Foo
{
	static
	{
		System.out.println("It is a static block");
	}
}

class DynamicLoading
```

```
{
	public static void main(String[] args)
	{
		try
		{
			Class.forName(args[0]);  //Command line argument
		}
		catch (ClassNotFoundException e)
		{
			System.err.println("Class is not available ....Plz check again");
		}
	}
}
```

java.lang.NoClassDefFoundError:-

It occurs when the class was present at the time of compilation but at runtime the required .class file is not available(mannually deleted by user) then we will get an exception i.e java.lang.NoClassDefFoundError.

```
class Hello
{
	void welcome()
```

```
	{
		System.out.println("Hello learner!!!!");
	}
}
class  NoClassDefFoundErrorDemo
{
	public static void main(String[] args)
	{
		Hello h = new Hello();
		h.welcome();
	}
}
```

Note :- delete the Hello.class file after compilation and then execute the program.

What is the drawback of new keyword :-

We know 'new' keyword is used to create the objcet but It demands the class name at the begning or at the time of compilation.

new keyword is not suitable to create the object for the classes which are coming at runtime.

In order to create the Object for the classes which are coming at runtime either from database or files, we should use newInstance() method available in class called Class.

```
class Student
{
}
class DynamicObjectCreation
{
	public static void main(String[] x) throws Exception
	{
		Object obj =  Class.forName(x[0]).newInstance();
		System.out.println("Object created for :"+obj.getClass().getName());
	}
}
```

```
class Student
{
	void message()  //instance method
	{
		System.out.println("Welcome students...");
	}
}
class DynamicObjectCreation
{
	public static void main(String[] x) throws Exception
	{
		Object obj =  Class.forName(x[0]).newInstance();  //Student
```

```
                Student st =(Student) obj;
                st.message();
        }
}
```

Program on class Class object, to get the details of the class

---

```
import java.lang.reflect.Method;
class ClassDescription
{
        public static void main(String[] args) throws Exception
        {
                int count = 0;
                 Class cls = Class.forName("java.lang.Integer");
                 System.out.println("Name of the class is :"+cls.getName());
                 System.out.println("Name of the package :"+ cls.getPackage());

    Method [] methods = cls.getDeclaredMethods();

                for(Method x : methods)
                {
                        count++;
                        System.out.println( x.getName());
                }
```

```
        System.out.println("Total methods are :"+count);
            }
}
```

---

2) Runtime data areas :-

Method area :-

In this area all class level information is available. Actually the .class file is dumpped here hence we have all kinds of information related to class is available like name of the class, name of the immediate super class, method and variable name, static variable and so on.

There is only one method area per JVM.

Heap memory :-

It stores information regarding object and instance variables. All the objects are created as a part of HEAP memory so automatically all the instance variables are also the part of HEAP memory.

The is only one HEAP area per JVM.

Stack area :-

For every thread, JVM creates a seperate runtime stack. Each stack is created as a part of stack memory. All the local variables are also the part of stack memory.

Each entry in the stack is called Stack Frame, Each stack frame containd three parts

1) Local variable Array

2) Frame Data

3) Operand Stack

Garbage collector :-

It is an automatic memory management in java. JVM internally contains a component called Garbage collector, It is responsible to delete the unused objects or the objects which are not containing any references in the memory.

Heap and stack diagram

OR

Grabage collector program

OR

Output of any complex program

```
class Customer
{
        private String name;
        private int id;

        public Customer(String name , int id)
        {
                this.name=name;
```

```
		this.id=id;
	}
	public void setId(int id)  //setter
	{
		this.id=id;
	}
	public int getId() //getter
	{
		return id;
	}
}
class Test
{
	public static void main(String[] args)
	{
		int val=100;
		Customer e = new Customer("Ravi",2);
		m1(e);
		//gc  //3000x object is eligible for garbage collector
		System.out.println(e.getId());
	}
	public static void m1(Customer e)
	{
		e.setId(5);
```

```
                e=new Customer("Rahul",7);
                e.setId(9);
                System.out.println(e.getId());
        }
}
```

```
public class Employee
{
        int id=100;
        public static void main(String[] args)
        {
                int val=200;
                Employee e1 = new Employee();
                e1.id=val;
                update(e1);

                System.out.println(e1.id);
    Employee e2 = new Employee();
                e2.id=500;

                switchEmployees(e2,e1);

                  //gc
```

```
                System.out.println(e1.id);
                System.out.println(e2.id);
        }

        public static void update(Employee e)
        {
    e.id=900;
            e=new Employee();
            e.id=400;
        }

        public static void switchEmployees(Employee e1,Employee e2)
        {
            int temp=e1.id;
            e1.id=e2.id;
            e2= new Employee();
            e2.id=temp;
        }
  }

---------------------------------------------------------------------------------

class Test1
{
        public static void main(String[] args)
```

```
	{
		public int x ;
		System.out.println(" x value is :"+x);
	}
}
```

//A local variable must be initialized as well as we can't use access modifier like public protected and private

---

```
class Demo
{
	int x = 100;
	public static void main(String[] args)
	{
		System.out.println("x value is :"+x); //error
	}
}
```

```
class Demo
{
	static int x = 100;
	public static void main(String[] args)
```

```
	{
		System.out.println("x value is :"+x);
	}
}
```

```
class Demo
{
	int x = 100;
	public static void main(String[] args)
	{
		Demo d1 = new Demo();
		System.out.println("x value is :"+d1.x);
	}
}
```

In order to access instance variable or instance nethod from a static method, we need to create an object first

```
class Foo
{
	int x = 100;//instance variable
	public static void main(String[] args)
	{
	    Foo f1 = new Foo();
		f1.m1();
```

```
                System.out.println(f1.x);
        }

        void m1()//instance method
        {
                System.out.println("It is m1 method");
        }
}
```

If a static method is available in the same class we can directly call the static method but if the static method is avaialble in another class then we need to call the static method with the help of class name

```
class Foo
{
        public static void main(String [] args )
        {
       m1();
                Sample.m2();
        }

        static void m1()
        {
                System.out.println("It is a static method...");
        }
```

```
}


class Sample
{
	static void m2()
	{
		System.out.println("static method inside m2");
	}
}
```

```
class Test
{
	int x; //instance variable


	void input(int x)  //instance nethod
	{
		this.x = x;
	}


	static void access()
	{
		System.out.println("x value is :"+x);
	}
}
```

```
class StaticAccess
{
	public static void main(String[] args)
	{
		Test t1 = new Test();
		t1.input(15);

		Test.access();
	}
}
```

```
class Test
{
	int x; //instance variable

	void input(int x)  //instance nethod
	{
		this.x = x;
		this.access();
	}

	 void access()
	{
		System.out.println("x value is :"+x);
```

```
	}
}
class StaticAccess
{
	public static void main(String[] args)
	{
		new Test().input(15);  //nameless Object OR Anonymous object


	}
}
```

```
class Test
{
	int x; //instance variable


	void input(int x)  //instance nethod
	{
		this.x = x;
	}


	 void access()
	{
		System.out.println("x value is :"+x);
	}
```

```
}

class StaticAccess

{

        public static void main(String[] args)

        {

           Test  t1 = new Test();

                    t1.input(15);

                    this.access();

        }

}
```

PC Register :-

In order to hold the current executing instruction of a thread we use PC register (Program Counter Register). For a single JVM we can have multiple PC Registers.

Native method stack :-

For every thread in java a seperate native stack is created. It stores the native method information.

Interpreter :-

JVM stands for Java Virtual Machine. It is a software in the form of Interpreter written in 'C'

language.

The main purpose of JVM to load and execute the .class file.JVM has a component called class loader subsystem responsible to load the required .class file as well as It allocates the necessary memory needed by the java program.

JVM executes our program line by line. It is slow in nature so java software people has provided a special compiler i.e JIT compiler to boost up the execution.

JIT compiler :-

It stands for just in time compiler. It is the part of JVM which increases the speed of execution of a java program.

It holds the frequently used instruction and make it available at the time of executing java program so the execution will become faster.

Working with class and Object :-

```
class Student
{
  private int sno;
  private String name;

  public void input()
```

```
        {
          sno =111;
          name = "Ravi";
        }

  public void show()
  {
    System.out.println(sno);
    System.out.println(name);
  }
  public static void main(String [] x)
  {
    Student s1 = new Student();
        s1.show();
                        s1.input();
                        s1.show();
  }
}

---------------------------------------------------------------------------------
class Student
{
  private int sno;
  private String name;
```

```
 Student()
          {
            System.out.println(sno);
            System.out.println(name);
          }

 public static void main(String [] x)
 {
   Student s1 = new Student();
 }
}
```

Note :- In this program from the first line of constructor we have a super call, which is calling the default constructor of Object class.

What happens internally when we create an Object ?

OR

When to declare instance variable and when to declare static variable?

//This following program describes whenever we create an object a seperate copy of instance variables will be created with different memory location.

```
class Test
{
	int x = 15;

	public static void main(String[] args)
	{
		Test t1 = new Test();
		Test t2 = new Test();

		++t1.x;	--t2.x;

		System.out.println(t1.x);
		System.out.println(t2.x);
	}
}
```

//static variables create the sible copy sharable by all the objects, so if there is a change by any object then it will reflect to all the objects

```
class Test
{
	static int x = 15;

	public static void main(String[] args)
	{
		Test t1 = new Test();
```

```
                Test t2 = new Test();


                ++t1.x;    ++t2.x;


                System.out.println(t1.x);

                System.out.println(t2.x);

        }

}

----------------------------------------------------------------------------------------class Student

{

  int rollNumber;

  String name;

  String address;

  static String collegeName = "Osmania";

}
```

We shoud declare instance variable if the value is different with each object on the other hand we should a variable as a ststic variable if the value is common for all the objcets like collegeName is a cooman value for all the Students.

The following explains about this concept.

```
class Student

        {

  int roll;

  String name;
```

```
  static String college ="JNTU"; //college name will be same for all students


  Student(int r,String n)
            {
  roll = r;
  name = n;
   }


void display ()
            {
            System.out.println(roll+" "+name+" "+college);
            }


 public static void main(String args[])
 {
 Student s1 = new Student (101,"Rahul");
 Student s2 = new Student (102,"Aswin");
 Student s3 = new Student (103,"Virat");
 s1.display();
 s2.display();
 s3.display();
 }
}
```

```
class Student
        {
  int roll;
  String name;
  static String college ="JNTU"; //college name will be same for all students

  Student(int r,String n)
        {
  roll = r;
  name = n;
   }

 void display ()
        {
        System.out.println(roll+" "+name+" "+college);
        }

 public static void main(String args[])
 {
 Student s1 = new Student (101,"Rahul");
 Student s2 = new Student (102,"Aswin");
 Student s3 = new Student (103,"Virat");
 s1.display();
 s2.display();
```

```
 s3.display();
 }
}
```

In the above program We have given together So, It is not an object oriented approach even we are creating the object. It increases tight coupling.

Why we pass parameter to a function :-

---

1) For geeting more information

2) To get the values from outside world

3) To initialize the instance variables of class.

How to define BLC and ELC classes in our Program

```
//BLC (Business Logic class)
class MyTeam
{
        int tno;
        String tname;

   public void input(int tno, String tname)
        {
```

```
        this.tno = tno;
                this.tname = tname;
        }

        public void show()
        {
                System.out.println("Number of member is :"+tno);
                System.out.println("Name of the member is :"+tname);
        }
}
class  Team   //ELC (Executable Logic class)
{
        public static void main(String[] args)
        {
            MyTeam mt = new MyTeam();
                mt.input(111,"Ravi");
                mt.show();
        }
}
```

What is the benefit of writing constructor in our program :-

If we write constructor in our program then compiler will not add any kind of constructor, we initialize our all instance veriables inside the constructor so variable initialization and variable

re-initialization both are done in the same place as shown in the diagram below.

Constructor :-

It is used to construct the Object that's why the name is constructor.

Constructor :-

It is a special method whose name is same as class name or in words we can say if the class name and method name both are same then it is called constructor.

The main purpose of constructor to initialize the object that initialize the instance variable of the class.

Every java class has a constructor either implicitly added by the compiler or explicitly written by the user.

A default constructor will be added by the compiler in a class to provide default values for the instance variables in case user has not written any kind of constructor.

Constructor never containing any return type including void also.

A constructor is automatically called and executed at the time of creating the object.

A constructor is called only once per object that means when we craete the object constructor

will be called and executed, if we create the object again then again the constructor will be called and executed.

Note :- A constrauctor can't be declared as static and final. A constructor can not be inherited in java.

Java supports 4 kinds of constructor

---

1) default constructor

2) parameterized constructor

3) copy constructor

4) private constructor

default constructor:-

If no argument is passed to the constructor then it is called default constructor

Program on default constructor :-

---

class Person

```
{
	int pid;
	String pname;

	Person()  //default constructor
	{
		pid = 111;
		pname = "Rahul";
	}

	public void show()
	{
		System.out.println("Person id is :"+pid);
		System.out.println("Person name is :"+pname);
	}
}

class PersonDemo
{
	public static void main(String[] args)
	{
		Person p1 = new Person();
		p1.show();
```

```
        Person p2 = new Person();
        p2.show();
    }
}
```

Program on parameterized constructor :-

If we pass one or more arguments to the constructor then it is called parameterized constructor.

```
class Person
{
    private String name;
    private int age;
    Person() //default constructor
    {
        name = "Rohit";
        age = 12;
    }
    Person(String name, int age) //parameterized constr.
    {
        this.name = name;
        this.age = age;
    }
```

```
	public void talk()
	{
		System.out.println("My name is : "+name);
		System.out.println("My age is : "+age);
	}
}

class ParameterizedDemo
{
	public static void main(String[] args)
	{
		Person p1 = new Person();
		Person p2 = new Person("Virat", 14);
		p2.talk(); p1.talk();
	}
}
```

Copy Constructor : -

If an object reference is passed to a constructor then it is called copy constructor.

The main purpose of copy constructor is to copy the contant of one object to another object.

Ex:-

```
class Student
{
  int roll;
  String name;

  Student() //default constructor
  {
  }

  Student(int r, String n) //parameterized constructor
  {
     roll = r;
     name = n;
  }

  Student(Student s)  //copy constructor
  {

  }

}
```

```
class Player
```

```
{
  String name1, name2;

  Player(String name1 , String name2)
  {
     this.name1 = name1;
              this.name2 = name2;
  }

  Player(Player p)
  {
    name1 = p.name2;
    name2 = p.name1;
  }
  public void show()
  {
   System.out.println(name1);
   System.out.println(name2);
  }
}

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

```
	{
		Player p1 = new Player("Virat", "Rohit");

		Player p2 = new Player(p1);

		p1.show();   p2.show();
	}
}
```

---

private constructor :-

We can declare a construor as private to make singletone class. In this class we can take static method which can be invoked without creating an object.

```
class Customer
{
	private Customer(){}

   static void m1()
	{
		System.out.println("m1 static method....");
	}

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

```
	{
		Customer c1 = new Customer(); //creating one object
		Customer.m1();
	}
}
```

Inheritance :-

Deriving a new class from existing class in such a way that the new class will acquire all the features and properties of existing class is called Inheritance.

It is one of the most imporatnt feature of oops which provides "CODE REUSABILITY".

In java we provide inheritance using a keyword called 'extends'.

Using inheritance mechanism the relationship between the classes is parent and child, According to C++ the parent class is called Base class and the Child class is called Derived class whereas According to Java the parent class is called as super class and the child class is called as sub class.

Using inheritance all the features of super class is bydefault available to the sub class so the sub class need not to start the process from begning onwards.

Inheritance follows top to bottom approach, In this hierarchy if we move towards upward direction more generalized properties will occur and on the other hand if we move towards downward direction more specialized properties will occur.

Inheritace provides two kinds of relationship

IS-A Relation :- It occurs between the classes.

HAS-A Relation :- It occurs between the class and its property.

```
import java.util.*;


class A
{
	int x,y;

	public void input()
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter the value of x :");
		x = sc.nextInt();

		System.out.print("Enter the value of y :");
		y = sc.nextInt();
	}
}

class B extends A
{
	public void show()
```

```
	{
		System.out.println("x value is :"+x);
		System.out.println("y value is :"+y);
	}
}

class SingleLevel
{
	public static void main(String[] args)
	{
		B b1 = new B();
		b1.input();
		b1.show();
	}
}
```

```
class Emp
{
	int eno;
	String ename;
	String eaddr;

	public void setEmp()
	{
```

```
                eno = 111;
                ename = "Virat";
                eaddr = "Delhi";
        }

   public void disEmp()
        {
      System.out.println("Employee Number is :"+eno);
                System.out.println("Employee Name is :"+ename);
                System.out.println("Employee Address is :"+eaddr);
        }
}

class Pemp extends Emp
{
        String dept;
        String desi;

        public void setPemp()
        {
                dept = "Cricket";
                desi = "Batsman";
        }
```

```
	public void disPemp()
	{
		System.out.println("Employee Department is :"+dept);
		System.out.println("Employee Designation is :"+desi);
	}
}
public class EmpDemo
{
	public static void main(String[] args)
	{
		Pemp p = new Pemp();
		p.setEmp();
		p.disEmp();
		p.setPemp();
		p.disPemp();
	}
}
```

Program on multi-level inheritance:-

```
class Student
{
	int rollNumber;
	String name;
```

```
	double fees;
}
class Science extends Student
{
	int phy, che;
}

class PCM extends Science
{
	int math;

	PCM(int rollNumber,String name, double fees, int phy, int che, int math)
	{
		this.rollNumber = rollNumber;
		this.name = name;
		this.fees = fees;
		this.phy = phy;
		this.che = che;
		this.math = math;
	}

	public void show()
	{
		System.out.println("Student Roll Number is :"+rollNumber);
```

```
                System.out.println("Student Name  is :"+name);
                System.out.println("Student Fees is :"+fees);
                System.out.println("Marks of Physics  is :"+phy);
                System.out.println("Marks of Chemistry is :"+che);
                System.out.println("Marks of Math is :"+math);
        }

        public void result()
        {
                int total = phy + che + math;
                System.out.println("Total marks is :"+total);
        }
}
public class StudentDemo
{
        public static void main(String[] args)
        {
                PCM p = new PCM(111,"Rahul",12000,89,88,90);
                p.show();
                p.result();
        }
}
```

Why java doesn't support multiple inheritance ?

By default Java does not support multiple inheritance because It is a situation where a sub class wants to inherit the properties of two or more super classes as a result It may produce ambiguity problem that is the reason java does not support multiple inheritance.

We have a concept called interface through which we can acheive multiple inheritance.

super keyword :-

It is used to access the member of immediate super class. In java we can use super keyword in 3 ways

1) To call the variable of the super class

2) To call the method of super class

3) To call the constructor of the super class***

To call the variable of super class :-

Whenever super class variable name and sub class variable name both are same and if we create an object of sub class then it will give more priority to its own class variable if we want to call super class variable then we should use super keyword.

super keyword always refers to its immediate super class as well as we should only use super keyword whenever the member of super class name and the member of sub class both are same.

```
class AA
{
        int x = 100;
}
class BB extends AA
{
         int x = 200;

         BB()
         {
                  System.out.println("Sub class x variable is :"+x);
                  System.out.println("Super class x variable is :"+super.x);

         }
}
public class SuperVar
{
        public static void main(String[] args)
        {
                 BB b1 = new BB();
        }

}
```

To call the super class method :-

Whenever the super class method name and sub class method name both are same and if we create an object of sub class then it will give more priority to its own class method.

If we want to invoke the super class method then we should use super

keyword.

```
class Super
{
	void show()
	{
		System.out.println("Super class show method...");
	}
}
class Sub extends Super
{
	void show()
	{
		super.show();
		System.out.println("Sub class show method...");
		super.show();
	}
}
public class SuperMethod
```

```
{
	public static void main(String[] args)
	{
		Sub s = new Sub();
		s.show();

	}

}
```

---

3) To call the constructor of super class :-

To call the super class constructor:-

Whenever we write a class in java and if we have not written any kind of constructor then automatically the compiler will add one default constructor to the class.

The first line of any constructor is reserved either for super() or this(). If a developer does not specify either super() or this() to the first line of constructor then compiler will automatically add super() to the first line of constructor.

super() :-

It is used to call the default constructor of the super class.

---

class Base

```
{
	Base()
	{
		System.out.println("Super class Constructor...");
	}
}
class Derived extends Base
{
	Derived()
	{
		System.out.println("Sub class constructor.....");
	}
}
public class ConstructorChaining
{
	public static void main(String[] args)
	{
		Derived d = new Derived();
	}
}
```

```
super("Naresh");
```

It is used to call parameterized constructor of the super class.

```
package com.ravi.veh;


class Parent
{
	Parent(String x)
	{
		System.out.println("Institute Name is :"+x);
	}
}
class Child extends Parent
{
	Child()
	{
		super("Naresh");
		System.out.println("Child class");
	}
}
public class ParameterizedDemoSuper
{
	public static void main(String[] args)
	{

		Child c = new Child();
```

```
	}


}
```

---

this() :-

It is used to call the default constructor of its own class

```
class Parent
{
	Parent()
	{

		System.out.println("Default constructor....");
	}
	Parent(String x)
	{
		this();
		System.out.println("Institute Name is :"+x);
	}
}
class Child extends Parent
{
	Child()
```

```
	{
		super("Naresh");
		System.out.println("Child class");
	}
}
public class ParameterizedDemoSuper
{
	public static void main(String[] args)
	{

		Child c = new Child();
	}

}
```

---

this("Ravi") :-

It is used to call the parameterized constructor of its own class

---

```
class Parent
{
	Parent()
	{
		this("Naresh");
```

```
		System.out.println("Default constructor....");
	}
	Parent(String x)
	{
		System.out.println("Institute Name is :"+x);
	}
}
class Child extends Parent
{
	Child()
	{
		System.out.println("Child class");
	}
}
public class ParameterizedDemoSuper
{
	public static void main(String[] args)
	{

		Child c = new Child();
	}

}
```

```
class A
{
	A()
	{
		System.out.println("Class A");
	}
}
class B extends A {


}


class C extends B
{
	C()
	{
		System.out.println("Class C");
	}
}
public class ConstructorTest
{
	public static void main(String[] args)
	{
		C c1 = new C();
```

```
	}
}
```

```
package com.ravi.veh;

class Shape
{
	int x;//2
	Shape(int x)
	{
		this.x = x;
		System.out.println("x value is :"+x);
	}
}
class Square extends Shape
{

	Square(int x)
	{
		super(x);
	}

	public void area()
```

```
	{
		System.out.println("The area of square is :"+(x*x));
	}
}
public class ShapeDemo
{
	public static void main(String[] args)
	{
		Square ss = new Square(2);   ss.area();
	}

}
---------------------------------------------------------------------------
class Shape
{
	int x;//2-> 4
	Shape(int x)
	{
		this.x = x;
		System.out.println("x value is :"+x);
	}
}
class Square extends Shape
{
```

```
	Square(int x) //2 -> 4
	{
		super(x);
	}

	public void area()
	{
		System.out.println("The area of square is :"+(x*x));
	}
}
class Rectangle extends Square
{
	int a;

	Rectangle(int l, int b)  //l=4	b=5
	{
		super(l);
		a = b;
	}

	public void area()
	{
		System.out.println("The area of Rectangle is :"+(x*a));
```

```
	}
}

public class ShapeDemo
{
	public static void main(String[] args)
	{
		Square ss = new Square(2);   ss.area();

		Rectangle rr = new Rectangle(4,5);   rr.area();
	}

}
```

Basic IQ:-

```
class Test1
{
	public static void main(String[] args)
	{
		int a = 010;     //octal 0-7
		int b = 0xadd; //Hexadecimal 0-f
		int c = 0b111; //Binary 0 and 1
```

```
                System.out.println("a value is :"+a);
                System.out.println("b value is : "+b);
                System.out.println("c value is : "+c);

        }
}
```

```
public class Test2
{
public static void main(String[] args)
        {
                short s = 135;
                byte b = (byte)s;
                System.out.println("Value is :"+b);
                System.out.println(Short.MAX_VALUE);
                System.out.println(Short.MIN_VALUE);

                long l = 123L;
                float f = l;
                System.out.println(f);

        }
}
```

```
---------------------------------------------------------------------------------

class Test3

{

	public static void main(String[] args)

	{

		double x = 015.29; //iN FLOATING LITERAL WE HAVE ONLY DECIMAL FORM, NO
OCTAL, HEXADECIMAL  AND BINARY


		 double y = 0167;


		 double z = 0187;//error


		System.out.println(x+","+y+","+z);

	}

}

---------------------------------------------------------------------------------

class Test4

{

	public static void main(String[] args)

	{

		double x = 0X29;


		double y = 0X9.15;


		System.out.println(x+","+y);
```

```
        }
}
--------------------------------------------------------------------------------
class Test5
{
        public static void main(String[] args)
        {
                double a = 0791; //error

                double b = 0791.0;

                double c = 0777;

                double d = 0Xdead;

                double e = 0Xdead.0;//error
        }
}
--------------------------------------------------------------------------------
class Test6
{
        public static void main(String[] args)
        {
          double a = 1.5e3;
```

```
            float b = 1.5e3; //error
            float c = 1.5e3F;
            double d = 10;
            int e = 10.0; //error
            int f = 10D;  //error
            int g = 10F; //error
          }
}
```

```
public class Test7
{
   public static void main(String[] args)
   {
          boolean c = 0; //error
      boolean d = 1; //error
      System.out.println(c);
      System.out.println(d);
   }
}
```

```
class Test8
{
          public static void main(String[] args)
```

```
        {
                boolean x = "true";
                boolean y = "false";
                System.out.println(x);
    System.out.println(y);
        }
}
```

```
class Test9
{
        public static void main(String[] args)
        {
                char c1 = 'A';
                char c2 = 65;
                char c3 = '\u0041';

                System.out.println("c1 = "+c1+", c2 ="+c2+", c3 ="+c3);
        }
}
```

```
class Test9
{
        public static void main(String[] args)
        {
```

```
                char c1 = 'A';

                char c2 = 65;

                char c3 = '\u0041';


                char c4 = '\u0061';


                System.out.println("c1 = "+c1+", c2 ="+c2+", c3 ="+c3+"c4="+c4);
        }
}
```

Note :- String is a final class in java

```
class Test10
{
        public static void main(String[] args)
        {
                String x = "india";
                x.toUpperCase();
                System.out.println(x);
        }
}
```

//== and equals(String x)

//== checks reference and equals() checks content

```
class Test11
{
	public static void main(String[] args)
	{
		String s1="naresh";
		String s2="naresh";
		String s3=new String("naresh");

    System.out.println(s1.equals(s2));//checks the content
		 System.out.println(s1.equals(s3));

	System.out.println(s1==s2); //check the reference but not content
		 System.out.println(s1==s3);
	}
}
```

```
//IQ
class Test12
{
	public static void main(String args[])
	{
		String s=15+29+"Ravi"+40+40;
		System.out.println(s);
```

```
                int x = 12; int y = 14;
                System.out.println("Sum of x and y is :"+x+y);
                System.out.println("Sum of x and y is :"+(x+y));
        }
}
```

While working with array , we have length variable whereas while working with String, we have length() method.

```
//IQ
class Test13
{
        public static void main(String args[])
        {
                String x = "India";
                System.out.println("it's length is :"+x.length);
        }
}
```

```
//IQ
class Test14
{
        public static void main(String args[])
        {
```

```
                String [] x = new String[3];
                System.out.println("it's length is :"+x.length());

        }
}
```

What is the difference among String, StringBuffer and StringBuilder

---

```
//Working with append() to describe the mutability
class Test15
{
public static void main(String args[])
        {
                StringBuilder sb1=new StringBuilder("Data");
                sb1.append("Base");
                System.out.println(sb1);

                StringBuffer sb2=new StringBuffer("Tata");
                sb2.append("Nagar");
                System.out.println(sb2);

                String sb3=new String("Data");
                sb3 = sb3.concat("Base");//DataBase
                System.out.println(sb3);
```

```
        }
}
```

Performance of StringBuffer and StringBuilder :-

```
long ms = System.currentTimeMilis();
```

```
//Program to demonstrate the performance of StringBuffer and StringBuilder classes.
class PerformanceProgram
{
        public static void main(String[] args)
        {
                long startTime = System.currentTimeMillis();
    StringBuffer sb = new StringBuffer("Java");
    for (int i=0; i<100000; i++)
                {
       sb.append("StringBuffer");
    }
                long endTime = System.currentTimeMillis();
    System.out.println("Time taken by StringBuffer: " + (endTime - startTime) + "ms");
```

```
                    //Starting time for StringBuilder
                    startTime = System.currentTimeMillis();
        StringBuilder sb2 = new StringBuilder("Java");
        for (int i=0; i<100000; i++)
                    {
           sb2.append("StringBuilder");
        }
        endTime = System.currentTimeMillis();
        System.out.println("Time taken by StringBuilder: " + (endTime - startTime) + "ms");
                    }
}
```

Operators :-

---

1) Binary operators :- It is also known as Arithmetic Operator. It works with two oprands that is the reason it is called Binary Operators.

Eg:

+ , - , *, / , %

Division Operator :-

While working with integeral literal when we divide a number by zero i.e (10/0 = Infinity and 0/0=Undefined) then there is no way to reprsent this Infinity and Undefined so in both the cases we will get java.lang.ArithmeticException and the execution sequence of our program will be

stopped.

On the other hand if we divide a number by 0.0 i.e floating point literal then java.lang.Float and java.lang.Double, both classes have been defined POSITIVE_INFINITY , NEGATIVE_INFINITY and NaN (Not a number) constants to represent positive infinity, negative infinity and undefined respectively.

Unary Operator :-

It works on single operand. we have 3 kinds of unary operator.

1) Unary minus(-)

2) Increment operator (++)

3) Decrement Operator (--)

Local varibale :-

The variables which are declared inside a method either as a method body or as a method parameter are called Local / Stack / Temporary / Automatic variable.

As per as its scope is concerned It can be accessible within the same method only.

A local variable must be initialized as well as we can't use access modifier (except final) on local variable.

Assignment Operator :-

It is used to assign right hand side value to left hand side. Both the values must be compatible with each other.

Relational Operator :-

These operators are used to compare the values. We have total 6 relational operator

1) > (Greater than)

2) >= (Greater than or equal)

3) < (less than)

4) <= (less than or equal)

5) == (double equal to)

6) != (Not equal to)

Note :- These operators always return boolean type.

If condition :-

It is used to check condition and based on the condition, it will execute the statement. The return type of if condition is always boolean value.

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

Nested if :-

---

if we place an if condition inside another if condition then it is called Nested if condition.

```
if(condition) //outer if
{
  if(condition) //Nested if condition (Inner if)
  {
  }
  else
  {
```

```
  }
}
else
{
}
```

Logical Operator :-

It is used to join or compound two or more statements in a single condition.

we have three kind of logical opertors

1) && AND Operator

2) || OR Operator

3) ! Not Operator

&& -> all the conditions must be true

|| -> at least one must be true

! -> It is an inverter so it makes true as a false and false as a true

Boolean Operators :-

These operators will work with boolean values.

& Boolean AND operator

| Boolean OR operator

! Boolean Not operator

& :- All the conditions must be true

| :- At least one condition must be true

! :- It is an inverter

Bitwise Operator :-

These operators are used to work with bit by bit operation. we have 3 kinds of bitwise operator

& Bitwise AND Operator

| Bitwise OR Operator

^ Bitwise X-OR Operator

& -> all conditions must be true

| -> at least one condition must be true

^ -> It will return true if both the arguments are alternate to each other.

29-12-2021

Bitwise Complement Operator :-

This operator does not work with booelan. It can work with only integer. It is represented by ~

Ternary Operator or Conditional operator (?:)

This operator is mainly used to reduce the size of if condition. It is called Ternary opertaor why because It uses 3 operators.

Member Access Operator(.) :-

This operator is represented by . (dot). It is used to access the member of the class like variables and methods.

new Operator :-

This operator is used to create the object. With the help of object referenec we can invoke the method or variable of the class.

If we have an instance variable or instance method (the non-static variable and method) then in order to call the method first of all we need to create an object.

instanceof operator :-

This operator is used to verify whether a reference variable is the instance of the particular object or not. If it is the instance of particular object then it will return true otherwise it will return false.

default values of the instance variable with class and object :-

---

Whenever we write a class in java and if we have not written any kind of constructor then one defualt constructor would be added by the compiler.

Now the main purpose of this constructor to initialize the instance variable of the class with some meaningful value called as default value.

int -> 0

float -> 0.0f

double -> 0.0d

char -> '\u0000'

String -> null

boolean -> false

```
//Arithmetic Operator
// Division Operator
class Test16
{
	public static void main(String [] args)
	{
		System.out.println(-10/0.0); //NEGATIVE_INFINITY
		System.out.println(10/0.0); //POSITIVE_INFINITY
		System.out.println(0/0.0);		//NaN

		System.out.println(10/0);  //ArithmeticException
		System.out.println(0/0);   //ArithmeticException
	}
}
```

Unary Operator :-

---

1) Increment Operator (++)

2) Decrement Operator (--)

3) Unary minus (-)

```
//*Unary Operators (Acts on only one operand)
//Unary minus Operator
class Test17
{
	public static void main(String[] args)
	{
		int x = 15;
		System.out.println(-x);
		System.out.println(-(-x));
	}
}
```

```
//Unary Operators
//Unary Pre increment Operator
class Test18
{
	public static void main(String[] args)
	{
		int x = 15;
		int y = ++x;
		System.out.println(x+":"+y);
	}
```

```
}
```

```
//Unary Operators
//Unary Post increment Operator
class Test19
{
	public static void main(String[] args)
	{
		int x = 15;
		int y = x++;
		System.out.println(x+":"+y);
	}
}
```

```
//Unary Operators
//Unary Pre increment Operator
class Test20
{
	public static void main(String[] args)
	{
		int x = 15;
		int y = ++15; //error
```

```
                System.out.println(y);
        }
}
```

```
//Unary Operators
//Unary Pre increment Operator
class Test21
{
        public static void main(String[] args)
        {
                int x = 15;
                int y = ++(++x); //error
                System.out.println(y);
        }
}
```

```
//Unary Operators
//Unary post increment Operator
class Test22
{
        public static void main(String[] args)
        {
                int x = 15;
                x++;
```

```
                System.out.println(x);
        }
}
```

```
public class Test28
{
static int i =5;
public static void main(String... args)
{
System.out.println(i++); //5
System.out.println(i); //6
System.out.println(++i);//7
System.out.println(++i+i++);  // 8 +8 = 16
}
}
```

```
class Test29
{
        public static void main(String[] args)
        {
                int x=2; int y=3;     //x=3
                if((y==x++) | (x< ++y))  //means if 3 is equal to 2 or 3<4
                {
```

```
                        System.out.println("x is :"+x+" y is :"+y);
                    }

            }
}
```

|| -> Logical Operator

| -> Boolean Operator

```
public class Test30
{
    public static void main(String[] argv)
    {
        int z = 5;
        if(++z > 5 || ++z > 6)
                                z++;
        System.out.println(z);

          }
}
```

```
public class Test31
{
    public static void main(String[] argv)
```

```
    {
        int val = 0;
        boolean test = (val == 0) || (++val == 2);
        System.out.println("test = " + test + "\nval = " + val);
    }
}
```

```
//Unary Operators
//Unary post increment Operator
class Test23
{
        public static void main(String[] args)
        {
                char ch ='A';
                ch++;
                System.out.println(ch);
        }
}
```

```
//Unary Operators
//Unary post increment Operator
class Test24
{
```

```
	public static void main(String[] args)
	{
		double d = 15.15;
		d++;
		System.out.println(d);
	}
}
```

```
//Unary Operators
//Unary Post decrement Operator
class Test25
{
	public static void main(String[] args)
	{
		int x = 15;
		int y = x--;
		System.out.println(x+":"+y);
	}
}
```

```
//IQ   --> max(int, type of i, type of j)
class Test26
```

```
{
	public static void main(String args[])
	{
		byte i = 1;
		byte j = 1;
		byte k = i + j;
		System.out.println(k);
	}
}
```

```
//IQ   --> max(int, type of i, type of j)
class Test27
{
	public static void main(String args[])
	{
		byte b = 6;
	  b = b + 7;
		System.out.println(b);


		byte c = 6;
	 c += 7;
	System.out.println(c);
```

```
        }
}
```

Polymorphism :-

It is Greek word whose meaning is "SAME OBJECT HAVING DIFFERENT BEHAVIOUR".

Poly = Many

morphism = Forms

void person(Walking)

void person(Running)

void person(Sleeping)

void person(Moving)

void person(Riding)

Polymorphism :-

Poly means "many" and morphism means "form". It is a Greek word whose meaning is "SAME OBJECT HAVING DIFFERENT BEHAVIOR"

In our real life a person or a human being can perform so many task similarly in our programming languages a method or a constructor can perform so many task.

Polymorphism can be divided into two categories :

1) Compile time Polymorphism OR Static Polymorphism

2) Run time Polymorphism OR Dynamic Polymorphism

Compile time polymorphism :-

The polymorphism which exist at the time of compilation is called static polymorphism. In static polymorphism compiler has a very good idea that which method is invoked depending upon the type of parameter we have passed in the method.

This type of preplan polymorphism is called static or compile time polymorphism.

Ex:- Method Overloading

Runtime polymorphism :-

The polymorphism which exist at runtime is called dynamic polymorphism. In dynamic polymorphism, compiler does not have any idea that which method is invoked, at runtime only the JVM will decide that which method is invoked depending upon the class type.

This type of polymorphism is called dynamic or Runtime polymorphism Or dynamic method dispatch.

Eg:- Method Overriding

Method Overloading :-

Writing two or more methods in the same class in such a way that the method name must be same and argument must be different is called method overloading.

While working with method overloading we can change the return type of the method.

```
//program on constructor Overloading
class Add
{
	Add(int a, int b)
	{
		System.out.println("Sum of two numbers :"+(a+b));
	}

	Add(int a, int b, int c)
	{
		System.out.println("Sum of three numbers :"+(a+b+c));
	}

	Add(float a, float b)
```

```
        {
    System.out.println("Sum of two floats are :"+(a+b));
        }
}
class StaticPoly
{
        public static void main(String[] args)
        {
                Add a1 = new Add(1,3,6);
                Add a2 = new Add(1,7);
                Add a3 = new Add(2.3f, 4.2f);
        }
}
```

```
//Program on method overloading
class Addition
 {
  public int add(int a1,int a2)
   {
      int a3 = a1+a2;
      return a3;
   }
  float add(float f1,float f2)
   {
```

```
    float f3 = f1+f2;
    return f3;
    }
}
class MethodOverload1
{
     public static void main(String args[])
    {
    Addition a = new Addition();
     int x = a.add(12,34);
     float y =a.add(2.45f,2.11f);
            System.out.println("Addition of two int is :"+x);
            System.out.println("Addition of two float is :"+y);
    }
}
```

While working with method Overloading if the same literal can assign to 2 or more methods then it will give more priority to the nearest data type Or in other words child priority is more than parent priority

```
class Test
{
        public void check(int b)
        {
                System.out.println("int executed");
```

```
	}
	public void check(long s)
	{
		System.out.println("long executed");
	}
}
class OverLoadCheck1
{
	public static void main(String[] args)
	{
		Test t = new Test();
		t.check(37);
	}
}
```

```
class Test
{
	public void check(String b)
	{
		System.out.println("String executed");
	}
	public void check(Object s)
	{
		System.out.println("Object executed");
```

```
	}
}
class OverLoadCheck2
{
	public static void main(String [] args)
	{
		Test t = new Test();
		t.check("Naresh");
	}
}
```

---

var-args :-

It stands for variable arguments. It is actually an array variable which can hold 0 parameter to n number of parameter of same type.

It is represented by ... (exactly 3 dots).By using this var-args conecpt, now we need to define only one method body for accepting different kinds of parameter.

var-args must be only one argument as well as last argument in the method otherwise there will be a compilation error.

```
class Test
```

```
{
	public void input(int ...a)   //var-args (Variable Argument)
	{
		System.out.println("Executed...");
	}
}
class VarArgs
{
	public static void main(String... args)
	{
		Test t = new Test();
		t.input();
		t.input(5);
		t.input(5,10);
		t.input(5,10,20);
	}
}
```

```
//Program to add the parameters value
class Test
{
	public void input(int ...a)
	{
```

```
                int total = 0;


                for (int b : a )
                {
        total = total + b;
                }
                System.out.println("Sum of parameter is :"+total);
          }
}
class VarArgs1
{
          public static void main(String... args)
          {
                Test t = new Test();
                t.input();
                t.input(5);
                t.input(5,10);
                t.input(5,10,15);
                t.input(5,10,15,20);
          }
}
```

//var args must be last and only one parameter

```
class Test
{
	public void input(int x, int y, int... a)
	{
		for (int z : a )
		{
			System.out.println(z);
		}
	}
}
class VarArgs2
{
	public static void main(String... args)
	{
		Test t = new Test();
		t.input(5,10,15,20,30,40);
	}
}
```

Method Overriding :-

Writing two or more methods in super and sub class in such a way that method signature

(Method name along with parameter) must be same is called Method Overriding.

Method Overriding is only possible with inheritance, if there is no inheritance there is no method overriding.

Generally we can't change the return type of the method while overriding a method but from JDK 1.6 onwards we can change the return type of the method by using a concept called Co-variant.

```
class Animal
{
	 void eat()
	{
		System.out.println("I can't say");
	}
}
class Dog extends Animal
{
	 void eat()
	{
		System.out.println("Non-Veg");
	}
	void bark()
	{
```

```
                System.out.println("BHU-BHU");
        }
}
class Horse extends Animal
{
        void eat()
        {
                System.out.println("Veg type");
        }
}
class AnimalDemo
{
        public static void main(String[] args)
        {
                Animal a = new Dog();
                a.eat();

                Animal a1 = new Horse();
                a1.eat();

        }
}
```

@Override Annotation :

It is a new Feature introduced in java from jdk 1.5 onwards. It is different from comment beacuse it ensures the compiler as well as User that the method is an overriden method and It must be available in the super class.

If we use @Override annotation and If we are not overriding the method from super class then compiler will generate an error.

```
class Super
{
  void show()
  {
    System.out.println("Super class show method...");
  }
}
class Sub extends Super
{
   @Override
  void show()
  {
            System.out.println("Sub class show1 method...");
  }
}
```

```
class  AnnotationDemo
{
	public static void main(String[] args)
	{
		Super s1 = new Sub(); //Upcasting
		s1.show();

		Sub s2 = (Sub) s1; //Downcasting
		s2.show();
	}
}
```

Role of access modifer while overridng a method :-

Method Overriding is only possible with inheritance, if there is no inheritance there will not be method overriding so private accees modifer is allowed at overriding.

While overriding a method the access modifier of sub class method must be greater or equal to the access modifer of super class method.

The following statement described the access modifer from greater to less

public > protected > default

(public is greater than protected, protected is greater than default)

```
--------------------------------------------------------------------------------
class RBI
{
	protected void loan()
	{
		System.out.println("Bank should provide loan...");
	}
}
class SBI extends RBI
{
	@Override
	 void loan()
	{
		System.out.println("Providing loan @ 9.0 %");
	}
}


class RBIDemo
{
	public static void main(String[] args)
	{
	   RBI r1 = new SBI();
		r1.loan();
```

```
	}
}
```

Note :- If super class method does not throw any exception using throws keyword then at the time of overriding the method the sub class method also should not throw the exception.

```
class MyThread extends Thread
{
	@Override
	public void run() throws InterruptedException
	{
		Thread.sleep(1000);
	}
}

class  ThreadDemo
{
	public static void main(String[] args)
	{
		MyThread m1 = new MyThread();
			m1.start();
	}
}
```

Co-variant in Java :

In general we can't change the return type of the method while overriding a method.

From JDK 1.6 onwards Java software people has provided a new concept called Co-variant through which we can change the return type of the method.

We can change the return type of the method using co-variant , co-variant describes the return type of the method must be in inheritance relationship.

The following program describes we can't change the return type of the method while overriding a method.

```
class Shape
{
	void draw()
	{
		System.out.println("Don't know about the shape...");
	}
}
class Rect extends Shape
{
	@Override
	int draw()
	{
    System.out.println("Drawing Rectangle....");
```

```
        return 0;
    }
}
class CoVariant
{
    public static void main(String[] args)
    {
        Shape s1 = new Rect();
        s1.draw();
    }
}
```

The following describes we can change the return type of the method while overriding by implementing Co-variant concept

```
class Shape
{
    Object draw()
    {
        System.out.println("Don't know about the shape...");
        return null;
    }
}
class Rect extends Shape
{
```

```
        @Override
        java.util.Date draw()
        {
    System.out.println("Drawing Rectangle....");
          return null;
        }
}
class CoVariant
{
        public static void main(String[] args)
        {
                Shape s1 = new Rect();
                s1.draw();
        }
}
```

```
class Test
{
        @Override
  public String toString()
        {
          return "Naresh Technology";
        }
```

```
	public static void main(String[] args)
	{
		Object t1 = new Test();
		System.out.println(t1);   //toString() of Object class
	}
}
```

---

final keyword :- (To provide some restriction)

To declare a class as a final :-

We can't inherit final class. When the class composition is very important and if we don't want the share the features and properties of class then we should declare a class as a final class.

If we declare a class as a final its variables can be modified, only the class behaviour is final.

```
 final class A
{
	private int x=100;
	public void setData()
	{
		x = 120;
		System.out.println(x);
	}
```

```
}
class B extends A  //error
{
}
public class FinalClassEx
{
	public static void main(String[] args)
	{
		B b1 = new B();
		b1.setData();
	}
}
```

To declare a method as a final :

If a method is declared as final we can't override the method the in the sub class. We should declare a method as final when we don't want to change the method body in the sub class method implementation.

```
class A
{
	int a = 10;
	int b = 20;
    final void calculate()
	 {
		int sum = a+b;
```

```
                System.out.println("Sum is :"+sum);
        }
}
class B extends A
{
        @Override
        void calculate() //error
        {
                int mul = a*b;
                System.out.println("Mul is :"+mul);
        }
}
class FinalMethodEx
{
        public static void main(String[] args)
        {
                A a1 = new B();
                a1.calculate();
        }
}
```

To declare a field (variable) as a final :

We cannot perform re-assignment to a final variable. In variable we write by uppercase letter in java.

```
class A
{
	final int A = 10;

	public void setData()
	{
		A = 10; //error re-assignment is not possible
		System.out.println("A value is :"+A);
	}
}
class FinalVarEx
{
	public static void main(String[] args)
	{
		A a1 = new A();
		a1.setData();
	}
}
```

Abstract class and abstract :-

A class that does not provide complete implementation (partial implementation) is defined as an abstract class.

An abstract method is a common method which is used to provide easiness to the programmer because the programmer faces complexcity to remember the method name.

An abstract method observation is very simple because every abstract method contains abstract keyword, abstract method does not contain any body and at the end there must be a terminator i.e ; (semicolon)

In java whenever action is common but implementations are different then we should use abstract method, Generally we declare abstract method in the super class and its implementation must be provided in the sub class.

if a class contains atleast one method as an abstract method then we should compulsory declare that class as an abstract class.

Once a class is declared as an abstract class we can't create an object for that class.

All the abstract methods declared in the super class must be overridden in the child class otherwise the child class will become as an abstract class hence object can't be created for the child class as well.

*An abstract class may or may not have an abstract method but an abstract method must have abstract class.

```
---------------------------------------------------------------------------

abstract class Shape

{

        abstract void draw();

}

class Rectangle extends Shape

{

        @Override

        void draw()

        {

          System.out.println("Drawing Rectangle.....");

        }

}


class Square extends Shape

{

        @Override

        void draw()

        {

                System.out.println("Drawing Square....");

        }

}


class Circle extends Shape
```

```
{
        @Override
        void draw()
        {
                System.out.println("Drawing Circle....");
        }
}

public class ShapeExample
{
        public static void main(String[] args)
        {
                Shape s ;

                s= new Rectangle();
                s.draw();

                s = new Square();
                s.draw();

                s = new Circle();
                s.draw();
        }
}
```

```
--------------------------------------------------------------------------------
abstract class Car
{
	int speed = 60;

	Car()
	{
		System.out.println("Constructor.....");
	}

	void getDetails()
	{
		System.out.println("It has 4 wheels...");
	}

	abstract void run();
}



class Naxon extends Car
{

	@Override
	void run()
```

```
        {
          System.out.println("Running safely...");
        }
}
public class IQ
{
        public static void main(String[] args)
        {
                Car c = new Naxon();
                System.out.println("Speed of the car is :"+c.speed);
                c.getDetails();
                c.run();

        }

}
```

Note :- If we have constructor in abstrct class then the constructor will be executed by internal super() call.

```
abstract class AA
{
        abstract void show();
        abstract void demo();
}
abstract class BB extends AA
```

```
{
        @Override
        void show()  //demo();
        {
          System.out.println("Show method implemented in the sub class...");
        }
}
class CC extends BB
{
        @Override
        void demo()
        {
          System.out.println("Demo method implemented in the sub class....");
        }
}
public class AbstractDemo
{
        public static void main(String[] args)
        {
                CC c1 = new CC();
                c1.show();
                c1.demo();
        }
```

```
}
--------------------------------------------------------------------------------
abstract class Shape
{
        abstract void input();
        abstract void area();
        Scanner sc = new Scanner(System.in);
}

class Rectangle extends Shape
{
        int l,b;

        @Override
        void input()
        {
          System.out.print("Enter the value of l :");
          l = sc.nextInt();

          System.out.print("Enter the value of b :");
          b = sc.nextInt();

        }
```

```
        @Override
        void area()
        {
                int area = l*b;
                System.out.println("The area of rectangle is :"+area);
        }

}
class Square extends Shape
{
   int a;
        @Override
        void input()
        {
                System.out.print("Enter the value of a :");
                 a = sc.nextInt();
        }

        @Override
        void area()
        {
                int area = a*a;
                System.out.println("The area of Square is :"+area);
        }
```

```
}

public class ShapeDemo
{

	public static void main(String[] args)
	{
		Shape s1 ;
		s1 = new Rectangle();
		s1.input();
		s1.area();

		s1 = new Square();
		s1.input();
		s1.area();
	}

}
```

07-FEB-22

Some important points to remember :-

---

1) An abstract method can't be declare private, final and static.

2) An abstract method can't be declare as synchronized.

3) static variables and static methods we can declare inside the abstract class.

4) we can declare constructor, variables and general method inside a abstract class.

Interface

---

interface (Upto 1.7) :-

An interface is a keyword in java which is similar to a class.

Upto JDK 1.7 an interfcae contains only abstract method that means there is a gurantee that inside an interfcae we don't have concrete methods.

In order to implment the member of an interface, java software people has provided implements keyword.

All the methods declared inside an interface is bydefault public and abstract so at the time of overriding we should apply public access modifier to sub class method.

All the variables declared inside an interface is bydefault public, static and final.

We should override all the methods of interface to the sub class otherwise the sub class will

become as an abstract class hence object can't be created.

We can't create an object for interface, but reference can be created.

By using interfcae we can acheive multiple inheritance in java.

```
public interface Moveable
{
  int SPEED = 80; //public + static + final

  void move();
}
class Naxon implements Moveable
{
        @Override
        public void move()
        {
         //SPEED = 100;          error
                System.out.println("The speed of the Naxon is :"+SPEED);
        }
}

public class Car {
```

```
	public static void main(String[] args)
	{
		Moveable m = new Naxon();
		m.move();
		System.out.println("Speed id :"+Moveable.SPEED);//static

	}

}
```

```
interface Client
{
	void sum();
	void sub();
	void mul();
}
class Developer implements Client
{
	int x, y;

	public void input()
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the value of x :");
```

```
        x = sc.nextInt();

        System.out.println("Enter the value of y :");
        y = sc.nextInt();
}

@Override
public void sum()
{
        int z = x + y;
        System.out.println("The sum is :"+z);

}

@Override
public void sub()
{
        int z = x - y;
        System.out.println("The sub is :"+z);

}

@Override
public void mul()
```

```
	{
		int z = x * y;
		System.out.println("The mul is :"+z);
	}
}



public class ClientDemo
{
  public static void main(String[] args)
  {
	Developer d = new Developer();
	d.input(); d.sum(); d.sub(); d.mul();
  }
}
```

```
interface i1
{
	void m1();
}
interface i2
{
	void m1();
}
```

```
class Implementation implements i1,i2
{
	@Override
	public void m1()
	{
		System.out.println("Multiple inheritance is possible...");
	}
}

public class MultipleInheritance
{

	public static void main(String[] args)
	{
		Implementation i = new Implementation();
		i.m1();
	}

}
```

08-02-2022

---

default method inside an interface :-

---

interface (JDK 1.8 onwards) :

From JDK 1.8 onwards java compiler allowed to write static and default method inside an interface.

default method we can write inside an interface so we can provide specific implementation for the classes which are implmenting from interface why because we can override default method inside the sub classes.

```
public interface Vehicle
{
	void move();

	void horn();

	default void digitalMeter()
	{
		System.out.println("Digital Meter");
	}
}

public class MyCar implements Vehicle
{
	@Override
```

```
	public void move()
	{
	 System.out.println("Moving with car...");
	}

	@Override
	public void horn()
	{
		System.out.println("Pop-Pop");
	}

	public void digitalMeter()
	{
	  System.out.println("New Version of car is running with digital meter..");
	}

}

public class MyBike implements Vehicle
{
	@Override
	public void move()
	{
		System.out.println("Moving with Bike");
```

```
	}

	@Override
	public void horn()
	{
		System.out.println("Peep-Peep");
	}

}

public class MainDemo {

	public static void main(String[] args)
	{
		Vehicle v;

		v = new MyCar();
		v.move();
		v.horn();
		v.digitalMeter();

		v = new MyBike();
		v.move();
```

```
                v.horn();


        }
}


----------//default method for specific class method implementation
interface HotDrink
{
        void prepare();


        default void expressPrepare()     //possible from jdk 1.8
        {
    System.out.println("Preparing with premium");
        }


}
class Tea implements HotDrink
{
        @Override
        public void prepare()
        {
                System.out.println("Preparing Tea");
        }
        @Override
```

```
        public void expressPrepare()
        {
    System.out.println("Preparing premium Tea");
        }
}
class Coffee implements HotDrink
{
 @Override
  public void prepare()
        {
                System.out.println("Preparing Coffee");
        }
}
class DefaultMethod
{
        public static void main(String[] args)
        {
                HotDrink hk1 = new Tea();
                HotDrink hk2 = new Coffee();
                hk1.prepare();
                hk1.expressPrepare();
                hk2.prepare();
        }
}
```

---

static method :-

From JDK 1.8 onwards we can define a static method inside an interface, static method is used to provide common implementation for all the classes which are implementing the interface. Here we can provide some common message because we can't override static method.

static methods are bydefault not available to implementer classes we can use static methods with the help of interface only.

```
//static method implemntation common for all
interface HotDrink
{
	void prepare();

	static void expressPrepare()
	{
		System.out.println("Preparing with no sugar...");
	}
}
class Tea implements HotDrink
{
	public void prepare()
```

```
        {
                System.out.println("Preparing Tea");
        }
}
class Coffee implements HotDrink
{
  public void prepare()
        {
                System.out.println("Preparing Coffee");
        }
}
class StaticMethod
{
        public static void main(String[] args)
        {
                HotDrink hk1 = new Tea();
                HotDrink hk2 = new Coffee();
                HotDrink.expressPrepare();
                hk1.prepare();
      hk2.prepare();
        }
}
```

Note :- We can't write static block, instance block and constructor inside the interface

---

```
interface Vehicle
{
	static void move()
	{
		System.out.println("Static method of Vechile");
	}
}
class StaticMethod1 implements Vehicle
{
	public static void main(String[] args)
	{
	   Vehicle.move();
		move();//error
		StaticMethod1.move();//error
		StaticMethod1 sm = new StaticMethod1();
		sm.move();//error

	}
}
```

```
interface I1
{
```

```
        default void m1()
        {
                System.out.println("Default method of I1 interface...");
        }
}
interface I2
{
        default void m1()
        {
                System.out.println("Default method of I2 Interface...");
        }
}
class MyClass implements I1,I2
{
  public void m1()
        {
          System.out.println("m1 method of MyClass");
                I1.super.m1();
                I2.super.m1();
        }
}
class MultipleInheritance
{
        public static void main(String[] args)
```

```
        {
                MyClass m = new MyClass();
                m.m1();
        }
}
```

Anonymous class :-

Implementing an inetrface without the help of class is called Anonymous class concept in Java.

```
//Anonymous class
interface Vehicle
{
    void run();
}
public class Anonymous
{
        public static void main(String [] args)
        {
       Vehicle v = new Vehicle()
                        {
                                @Override
                                public void run()
                                        {
```

```
                                        System.out.println("Running Safely");
                                }
                        };
                        v.run();
                }
}
```

```
interface Vehicle
{
    void run();
}
public class Anonymous1
{
        public static void main(String [] args)
        {
                Vehicle car = new Vehicle()
                        {
                                @Override
                                public void run()
                                        {
                                                System.out.println("Running Car");
                                }
                        };
                        car.run();
```

```
                Vehicle bike = new Vehicle()
                {
                        @Override
                        public void run()
                                {
                                        System.out.println("Running Bike");
                        }
                };
                bike.run();
        }
}
```

```
class Anonymous2
{
  public static void main(String [] args)
   {
      Runnable r = new Runnable()
            {
                        @Override
          public void run()
                      {
```

```
                System.out.println("Running....");
            }
        };
        r.run();
    }
}
```

Lambda Expression :-

---

1) It is an anonymous function.

2) It is used to enable functional programming in java.

3) It can be used with functional interface only

4) It is used to concise the code as well as we can remove boilerplate code (duplicate code).

5) If the body of the lambda expression contains onle one statement then curly braces are optional.

6) We can also remove the type of the variable.

```
@FunctionalInterface
interface Moveable
```

```
{
        void move();
}
class Lambda1
{
        public static void main(String[] args)
        {
                Moveable m = () ->
                {
                System.out.println("Hello Welcome to Lambda Expression");
                };
                m.move();
        }
}
```

```
@FunctionalInterface
interface Calculate
{
        void add(int a, int b);
}
class Lambda2
{
        public static void main(String[] args)
        {
```

```
                Calculate c = (a,b)->
                {
                        System.out.println("Sum is :"+(a+b));
                };
                c.add(12,12);
        }
}
```

```
@FunctionalInterface
interface Length
{
         int getLength(String str);
}
class Lambda3
{
        public static void main(String[] args)
        {
                Length l = (str)-> {          return str.length(); };

                System.out.print("Length is :"+l.getLength("India"));
        }
}
```

```
@FunctionalInterface
```

```
interface Length
{
	int getLength(String str);
}

class Lambda4
{
	public static void main(String[] args)
	{
		Length l = (str)-> str.length();

		System.out.print("Length is :"+l.getLength("Ravi"));
	}
}
```

Predefined functional interfaces :-

In order to help the programmer to write concise code in dat to day programming, java software people has provided following predefined functional interfaces as a part of java.util.function.

1) Predicate

2) Consumer

3) Supplier

4) Function

Predicate<T>

It is a predefined functional interface through which we can verify one boolean argument value.

```
@FunctionalInterface
interface Predicate<T>
{
  public abstract boolean test(T t);
}
```

In general how to check whether a number is even or odd

```
public boolean test(Integer i)
{
  if(i % 2==0)
    return true;
   else
    return false;
}
```

Now How to write the same code in Lambda Expression

```
(Integer i) -> i % 2 == 0;
```

Even we can remove this Integer

```
i -> i%2 == 0;
```

```
import java.util.function.*;

class Lambda5
{
	public static void main(String[] args)
	{
      Predicate<Integer> p = i-> i%2 ==0;
			System.out.println(p.test(10));
			System.out.println(p.test(15));
	}
}
```

Exception Handling :-

An exception is an abnormal situation or an unexpected situation in a normal execution flow.

Due to an exception our execution of the program will be disturbed first and then terminated

permanently.

An exception occurs due to dependency, when one part of the program is dependent to another part in order to complete the task then there might a chance of getting an exception.

EXCEPTION ALSO OCCURS DUE TO THE WRONG INPUT GIVEN BY THE USER.

The following program explains that whenever an exception occurs the execution of the program will be terminated abnormally and Program will be in Halt mode.

```
public class Test
{
	public static void main(String[] args)
	{
	System.out.println("main method started");
	int a = 10;
	int b = 0;
	int c = a/b;		//Program will halt and terminated abnormally
	System.out.println("c value is :"+c);
	System.out.println("main method ended");

	}
}
```

Exception Hierarchy :

As a programmer we are responsible to handle the exception, we are not responsible to handle the error. System Admin is responsible to handle the error.

The Exceptions which occurs commonly in java are called Checked Exception.

The Exceptions which occurs rarely in java are called Unchecked Exception.

Errors and RuntimeExceptions comes under the category of Unchecked Exception.

Some basic criteria for Exception and its corrosponding classes :-

---

1) 10 /0 :- java.lang.ArithmeticException

2) int []a = {12,78,90};

    System.out.println(a[3]); :- ArrayIndexOutOfBoundsException

3) String x = "Ravi";

   int y = Integer.parseInt(x); :- NumberFormatException

4) String a = null;

   a.length();   :- NullPointerException

Write a program in java to proof that Exception is the super class of all the Exceptions we have

in java.

```
public class ExceptionSuper
{
	public static void main(String[] args)
	{
		Exception e1 = new ArithmeticException("User has given zero");
		System.out.println(e1);

		Exception e2 = new ArrayIndexOutOfBoundsException("Out of the limit");
		System.out.println(e2);
	}
}
```

```
 class ArithmeticException extends Exception
{
  ArithmeticException()
  {
  }

  ArithmeticException(String message)
  {
    super(message);
```

```
  }
}
```

The object orientation has provided some mechanism to handle the exception by using the following keyword :-

1) try

2) catch

3) finally

4) throw

5) throws

try :-

Whenever our statement is error suspecting statement then put that statement inside the try block.

The try block will trace the program line by line and if any exception occurs then It will create the exception object and throw the exception object to the nearest catch block.

try block must be followed either by catch block or finnaly block or both. In between the try-catch we can't write any kind of statement.

catch block :-

The main purpose of catch block to handle the exception which is thrown by try block.

The catch block will only execute if there is an exception inside the try block.

```
public class TryDemo {

        public static void main(String[] args)
        {
        System.out.println("Welcome User");
        try
        {
                int a = 10;
                int b = 0;
                int c = a/b;
                System.out.println("c value is :"+c);
        }
        catch(Exception e)
        {
           System.err.println(e);
      System.out.println("--------------------------------");
      e.printStackTrace(); //For complete details
      System.out.println("--------------------------------");
      System.out.println(e.getMessage());
        }
```

```
        System.out.println("main method ended");
        }
}
```

```
//Exception handling describes to provide user-friendly messages to our client
public class CustomerDemo
{
        public static void main(String[] args)
        {
                System.out.println("Welcome user!!!");
                try
                {
            int a = 10;
            int b = 0;
            int c = a/b;
            System.out.println("c value is :"+c);
                }
                catch(Exception e)
                {
                        System.out.println("Please Don't put zero");
                }
                System.out.println("Hello user your program is completed!!!");
        }
}
```

Multiple try-catch :-

Depending upon our requirements we can take multiple try catch block.

```
public class MultipleTryCatch
{
	public static void main(String[] args)
	{
		try
		{
		  int a=100,b=0,c;
		  c = a/b;  // new ArithmeticException();
		  System.out.println("c value is :"+c);
		}
		catch(ArithmeticException e)
		{
		 System.err.println("Divide by zero problem....");
		}
		try
		{
			int x[] = {12,90};
			System.out.println(x[2]);
		}
```

```
                catch(ArrayIndexOutOfBoundsException e)
                {
                  System.err.println("Array is out of limit...");
                }
                System.out.println("Main completed..");
        }
}
```

Nested try block :

In java it is possible to define nested try block i.e one try block inside another try block.

The inner try block will only execute if we don't have any exception in the outer try block.

We shouls use nested try block when the inner try block has some which depends upon the code written iside outer try block.

```
public class NestedTry
{
        public static void main(String[] args)
        {
                try//Outer try
                {
                        String x = "naresh";
                        System.out.println("The length of "+ x+ " is :"+x.length());
```

```
                    try //inner try
                    {
                       String y = "456";
                       int z = Integer.parseInt(y);
                       System.out.println("z value is :"+z);
                    }
                    catch(NumberFormatException e)
                    {
                      System.err.println("Number is not in a format");
                    }
               }
               catch(NullPointerException e)
               {
                    System.err.println("Null Pointer problem...");
               }
     System.out.println("Main is completed....");
         }
}
```

---

try with multiple catch block :

We should take multiple catch block for providing more clearity regarding the exception.

While working with multiple catch block with a single try the super class catch block must be

the last catch block.

```
public class MultyCatch
{
	public static void main(String[] args)
	{
		System.out.println("Main Started...");
		try
		{
			int a=10,b=0,c;
			c=a/b;
			System.out.println("c value is :"+c);

			int x[] = {12,78,56};
			System.out.println(x[4]);
		}
		catch(ArrayIndexOutOfBoundsException e1)
		{
			System.err.println("Array is out of limit...");
		}
		catch(ArithmeticException e2)
		{
			System.err.println("Divide By zero problem...");
		}
```

```
                catch(Exception e)
                {
                        System.err.println("General Catch block");
                }
                System.out.println("Main Ended...");
        }
}
```

---

finally :-

---

finally block in java :-

The finally used to handle the resources. According to software engineering the resources are memory creation, buffer creation, Opening of a databade, opening of a file and so on, So we need to handle them carefully.

finally is a block which is guranted for execution whether an exception has been thrown or not.

We should write all the closing statements inside the finally block so finally block will execute and close all the resources.

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

```
                try
                {
                        System.out.println("main method started");
                        int a = 10;
                        int b = 0;
                        int c = a/b;  // new ArithmeticException();  HALT
                        System.out.println("c value is :"+c);
                }
                finally
                {
                        System.out.println("Finally block will be executed....");
                }
        System.out.println("Program is halt so It will not be executed");
           }
}
```

Note :- If we write try with finally only the resources will be handled but not the exception whereas if we write try-catch and finally then exception and resources both will be handled.

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

```
                System.out.println("Main is started!!!");
                int a[] = {12,67,56};
                System.out.println(a[3]);
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.err.println("Exception is handled here..");
            }
            finally
            {
              System.out.println("Resources will be handled here!!");
            }
            System.out.println("Main method ended!!!");
        }
}
```

Exception propagation :-

Whenever we call a method and if the the callee method contains any kind of exception and if callee method doesn't contain any kind of exception handling mechanism then it will propagate the exception to caller method for handling purpose. This is called Exception Propagation.

If the caller method does not contain any exception handling mechanism then JVM will terminate the method from the stack frame hence the remaining part of the method(m1 method) will not be executed even if we handle the exception in another caller method like main.

If all the caller method does not contain any exception handling mechanism then exception will be handled by JVM, JVM has default exception handler which will provide the exception message and terminates the program abnormally.

```
public class ExceptionPropagation
{
        public static void main(String[] args)
        {
           System.out.println("main started!!!");
           try
            {
                    m1();
            }
            catch(Exception e)
            {
      System.out.println("Handled in main!!!");
            }
           System.out.println("main Ended!!!");
        }

        static void m1()
        {
                System.out.println("m1 started!!!");
```

```
            m2();
        System.out.println("m1 Ended!!!"); //This line will not be executed
    }

    static void m2()
    {
            System.out.println(10/0);
    }
}
```

---

return statement with try-catch and finally

If we have return statement inside try block then catch block should also contain return statement otherwise there will be a compilataion error.

It is always better to write return statement in finally so need to define return statement inside try and catch.

After finally we should not write anything otherwise code will become unreachable.

```
public class ReturnExample
{
   public static void main(String[] args)
   {
      System.out.println(methodReturningValue());
```

```
    }
        @SuppressWarnings("finally")
        static int methodReturningValue()
    {
       try
       {
          System.out.println("Try block");
          System.out.println(10/0);



       }
       catch (Exception e)
       {
          System.out.println("catch block");


       }
       finally
       {
           System.out.println("Finally block");
          return 15;
       }

    }
}
```

Checked vs Unchecked exception :-

Checked Exception :

In java some exceptions are very common exceptions are called Checked excption here compiler take very much care and wanted the clearity regarding the exception by saying that by using this code you may face some problem at run time and you did not explain me how would you handle this situation at runtime are called Checked exception, so provide either try-catch or declare the method as throws.

Unchecked Exception :-

The exceptions which are rarely occurred in java and for these kinds of exception compiler does not take very much care are called unchecked exception.

Unchecked exceptions are directly entertain by JVM because they are rarely occurred in java.

----------When to provide try-catch or declare the method as throws :-

---

We should provide try-catch if we want to handle the exception by own as well as if we want to provide user-defined messages to the client but on the other hand we should declare the method as throws if we are not interested to handle the exception and try to send it to the JVM for handling purpose.

---

what is difference between throw and throws :-

---

throw :

In case of predefined exception try block is responsible to create the exception object and throw the exception object to the nearest catch block but it works with predefined exception only.

If a user wants to throw an exception based on his own requirement and specification then we should write throw keyword to throw the user defined exception object.

throws :-

In case of checked Exception if a user is not interested to handle the exception and try to throw the exception to JVM, try to skip from the current situation then we should declare a method as throws.

Note :- If a userdefined exception, we extend our userdefined exception class from RuntimeException then we need not to put try-catch or declare the method as throws for "throw keyword statement".

On the other hand, if we extend our userdefined exception class from Exception then compulsory we need to write "throw keyword statement"

in the protection of try-catch or we should declare the method as throws otherwise we will get a compilatioin error.

Note :-

In case of checked exception, handling the exception is compulsory because we deal with some

other resources so compiler wanted some clearity.

Note :- In comparison to Statement interface if we use PreparedStaement interface then it will reduced the overhead at the data base because in case of PreparedStaement interface the database engine has prepare the "query plan" only once.

How to create userdefined excetion :

---

1) Checked Exception :- class UDE extends Exception

Note :- try-catch or throws is compulsory

```
//Program
class InvalidAgeException extends Exception
{
 InvalidAgeException(String s)
 {
  super(s);
 }
}
class CustomExceptionWithChecked
            {
  static void validate(int age)
  {
    try
```

```
    {
                        if(age<18)
     throw new InvalidAgeException("Age not valid");
    else
     System.out.println("welcome to vote");
    }
    catch (Exception e)
    {
                         System.out.println(e);
    }
  }

  public static void main(String args[])
   {
       validate(17);
   }
}
```

---

2) Unchecked Exception :- class UDE extends RuntimeException

```
class GreaterMarksException extends RuntimeException
{
            GreaterMarksException(){}
```

```
        GreaterMarksException(String message)
        {
                super(message);
        }
}
public class CustomUncheckedException
{
        public static void main(String[] args)
        {
                validateMarks(102);
        }
        static void validateMarks(int marks)
        {
    try
    {
                        if(marks > 100)
        throw new GreaterMarksException("Marks is Invalid");
      else
        System.out.println("Your marks is :"+marks);
    }
    catch (Exception e)
    {
                        System.out.println(e);
```

```
        }
            }
}
```

Thread

Processes are heavy weight.

Threads are light weight.

Thread :-

A thread is the basic unit of CPU which can run with another thread at the same time within the same process.

It is well known for independent execution. The main purpose of multithreading to boost the execution sequence.

How to create Threads in Java :

There are two ways to create Thread in java

1) By extending Thread class

2) By implementing Runnable interface

Note :- Thread is a predefined class available in java.lang package, on the other hand Runnable is a prdefined functional interface inside java.lang package.

start() :-

It is a predefined method of Thread class and it will perform following two tasks

1) It will make a request to the Operating System to assign a new Thread for the program

2) It will internally invoke the run();

```
class MyThread  extends Thread
{
	@Override
	public void run()
	{
		System.out.println("Child thread is running...");
	}
}
public class UserThread
{
	public static void main(String[] args)
	{
		System.out.println("Main thread is running..");
		MyThread mt1 = new MyThread();
		mt1.start();//New thread is created here
```

```
                System.out.println("Main thread ended");
        }
}
```

Conclusions :

---

1) MyThread mt1 = new MyThread();

It will create new state of Thread i.e New state(Born state) but the Thread has not started yet

2) mt1.start() :

Thread is alive now, even though the run() is not started to execute the content of run(). It performs the following task

a) A new thread of execution starts with a sepearte Stack

b) The Thread moves from new state(born state) to runnable state(ready to run)

c) When a thread will get a chance to run() by an order of Thread Schedular then it will execute its target run() method.

d) Now, after complete execution of run() method thread will become Dead and it cann't be re-

started, if we try to restart it will generate java.lang.IllegalThreadStateException

3) public void run()

{

}

The user or chile Thread is eeecuting its own run method, after successful execution of run() method, the Thread is considered as Dead and we can't re-start the Thread.

4) MyThread mt1 = new MyThread();

mt1.run();

It is legal, but It does not start a new Thread

---

public boolean isAlive() :

It is a predefined method of Thread class through which we can find out whether a thread has started or not ?

As we know a thread starts after calling the start() so if we use isAlive() before start() method, it will return false but if the same isAlive() if we invoke after the start() method, it will return true.

We can't restart a thread in java if we try to restart then It will generate an exception i.e java.lang.IllegalThreadStateException.

```
 class Foo extends Thread
{
	@Override
	public void run()
	{
		System.out.println("Child thread is running...");
		System.out.println("It is running with seperate stack");
	}
}
public class IsAlive
{
	public static void main(String[] args)
	{
		System.out.println("Main method is started..");

		Foo f = new Foo();  //Thread has not started yet

		System.out.println(f.isAlive()); //false

		f.start();//new Thread is created

		System.out.println(f.isAlive());//true

		System.out.println("Main method is ended..");
```

```
   }
}
```

Anonymous class concept with Thread class :

```
public class AnonymousThread
{
public static void main(String args[])
{
        Thread t1 = new Thread()  //t1.start();
        {
                @Override
           public void run()
                {
                   System.out.println("My Thread One");
                }
        };

        Thread t2 = new Thread()
        {
                @Override
                public void run()
                {
```

```
                System.out.println("My Thread Two");
            }
        };
            t1.start();
            t2.start();
    }
}
```

Setting and getting the name of the thread :

The Thread class has provided two predefined methods setName(String name) and getName() to set and get the name of thread respectively.

If we don't set the name explicitly then bydefault the name of theread would be Thread-0, Thread-1, Thread-2 and so on.

//Program to get the name of the Thread

```
class Test extends Thread
{
    @Override
    public void run()
    {
        System.out.println(Thread.currentThread().getName()+" thread is running..."); //Thread-0, Thread-1
```

```
	}
}
public class ThreadName
{
	public static void main(String[] args)
	{
		Test t1 = new Test();
		Test t2 = new Test();
		t1.start();		//run();
		t2.start();		//run();
		System.out.println(Thread.currentThread().getName()+" thread is running...");
	}
}
```

---

setName(String name) :-

```
class Demo extends Thread
{
	@Override
	public void run()
	{
		System.out.println(Thread.currentThread().getName()+" thread is running...");
	}
}
```

```
public class ThreadName1
{
	public static void main(String[] args)
	{
	    Demo d1 = new Demo();
	    Demo d2 = new Demo();

	    d1.setName("Child1");
	    d2.setName("Child2");


	    d1.start();
	    d2.start();

	    System.out.println(Thread.currentThread().getName()+" thread is running...");
	}
}
```

Creating Thread by implements Runnable interface :

```
class Demo1 implements Runnable
{
  @Override
  public void run()
```

```
  {
    System.out.println("child thread");
  }
}
class RunnableDemo
{
 public static void main(String [] args)
 {
   Thread t1 = new Thread(new Demo1());
   t1.start();
  }
 }
```

In between extends Thread and implements Runnable which one is better?

In between extends Thread and implements Runnable approach, implements Runnable is more better due to the following reason

1) When we use extends Thread, all the methods and properties of Thread class is available to sub class so it is heavy weight but this is not the case while implementing Runnable interface.

2) As we know Java does not support multiple inheritance using classes so in the extends Thread approach we can't extend another class but if we use implments Runnable interface still we have chance to extend another class and we can also implement one or more interfaces.

Anonymous class concept with Runnable interface :

```
-------------------------------------------------------------

class AnonymousRunnable

{

	public static void main(String [] args)

	{

		Runnable r1 = new Runnable()

		{

					@Override

					public void run()

					{

						System.out.println("Runnable Demo1...");


					}


		};


		Runnable r2 = new Runnable()

		{

					@Override

					public void run()

					{

						System.out.println("Runnable Demo2...");


					}
```

```
                };

                Thread t1 = new Thread(r1);
                Thread t2 = new Thread(r2);

                t1.start();
                t2.start();
        }
}
```

Thread.sleep(long ms) :-

The main purpose of sleep() method to put a Thread into temporarly waiting state, the waiting period of the Thread will depend upon the parameter we passed inside the sleep() method, It takes long ms as a parameter.

It is a static method so we can directly call sleep() method with the help of class name.

It throws a checked exception i.e InterruptedException so we should write sleep() inside try-catch or declare the method as throws.

```
//program on sleep()
class Sleep extends Thread
{
```

```
	@Override
	public void run()
	{
		for(int i=1;i<=10; i++)
		{
			try
			{
				Thread.sleep(1000);
			}
			catch(InterruptedException e)
			{
				System.err.println("Thread has interrupted...");
			}
			System.out.println("i value is :"+i);
		}
	}
}
public class SleepDemo
{
	public static void main(String[] args)
	{
		System.out.println("Main Thread started..");
		Sleep s1 = new Sleep();
		s1.start();
```

```
	}
}
```

Note :- If we pass negative value to the sleep() method then it will throw an exception i.e java.lang.IllegalArgumentException.

```
package com.ravi.basic;


class MyTest extends Thread
{
	@Override
	public void run()
	{
		for(int i=1; i<=5; i++)
		{
			try
			{
				Thread.sleep(1000);
			}
			catch(InterruptedException e)
			{
				System.err.println("Thread has interrupted");
			}

			System.out.println(i+" by "+Thread.currentThread().getName());
```

```
                }
        }
}


public class SleepDemo1
{
        public static void main(String[] args)
        {
          System.out.println(Thread.currentThread().getName()+" thread");
                MyTest t1 = new MyTest();
                MyTest t2 = new MyTest();


            t1.start();  t2.start();
        }
}
```

In the above program we have all together 3 threads are created i.e main thread, Thread-0 and Thread-1

```
package com.ravi.basic;


class MyTest extends Thread
{
```

```
	@Override
	public void run()
	{
		for(int i=1; i<=5; i++)
		{
			try
			{
				Thread.sleep(1000);
			}
			catch(InterruptedException e)
			{
				System.err.println("Thread has interrupted");
			}

			System.out.println(i+" by "+Thread.currentThread().getName());
		}
	}
}



public class SleepDemo1
{
	public static void main(String[] args)
	{
```

```
                System.out.println(Thread.currentThread().getName()+" thread");
                MyTest t1 = new MyTest();
                MyTest t2 = new MyTest();



          t1.run();  t2.run();
        }
}
```

In the above program we have only thread i.e main thread because we don't have start() method.

---

join() method :-

The main purpose of join() method to put the parent thread into waiting state till the completion of child Thread.

It also throws checked exception i.e InterruptedException so better to use try catch or declare the method as throws.

It is an instance method so we can call this method with the help of Thread object reference.

```
//Program on join method
class Join extends Thread
{
```

```
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(Exception e) {}
                        System.out.println(i);
                }
        }
}
public class JoinThread
{
        public static void main(String args[]) throws InterruptedException
        {
                System.out.println("Main started");
                Join t1=new Join();
                Join t2=new Join();
                Join t3=new Join();

                t1.start();
```

```
                t1.join();          //main Thread will halt here


                t2.start();


                t3.start();


                System.out.println("Main Ended");
        }
}
```

Problem with multithreading :-

Multithreading is very good to complete our task as soon as possible but in some situation, It provides some wrong data or wrong result.

In Data Race or Race condition, all the threads try to access the resource at the same time so the result will be corrupted.

```
class Reserve implements Runnable
{
        int available = 1;
        int wanted;
        Reserve(int wanted)
```

```
        {
                this.wanted = wanted;
        }

        @Override
        public  void run()
        {
                System.out.println("Available berth ="+available);

                if(available >= wanted)
                {
                        String name = Thread.currentThread().getName();
                        System.out.println(wanted + " berth reserved for :"+ name);
                        available = available - wanted;
                }
                else
                {
                        System.out.println("Sorry!! No berth is available");
                }
        }

}
public class RailwayRservation
{
```

```
	public static void main(String[] args)
	{
		Reserve r = new Reserve(1);

		Thread t1 = new Thread(r);
		t1.setName("Rohit");

		Thread t2 = new Thread(r);
		t2.setName("Virat");

		t1.start(); t2.start();
	}

}
```

```
class MyThread implements Runnable
{
	String str;
	MyThread(String str)
	{
		this.str=str;
	}
	@Override
	public void run()
```

```
        {
                for(int i=1; i<=10; i++)
                {
                        System.out.println(str+ " : "+i);
                        try
                        {
                                Thread.sleep(100);
                        }
                        catch (Exception e)
                        {
                                e.printStackTrace();
                        }
                }
        }
}
public class Theatre
{
        public static void main(String [] args)
        {
        MyThread obj1 = new MyThread("Cut the Ticket");
        MyThread obj2 = new MyThread("Show the Seat");

                Thread t1 = new Thread(obj1);
                Thread t2 = new Thread(obj2);
```

```
                t1.start();

                t2.start();

        }

}
```

Synchronization in java :-

Synchronization is a technique through which we can control multiple threads but accepting only ONE THREAD AT ANY TIME.

In java we can acheive synchronization by using synchronized keyword.

It can be divided into two types

1) Method level synchronization

2) Block level synchronization

Method level synchronization :-

In method level synchronization, the entire method gets synchronized so all the thread will wait at method level and only one thread will enter inside the synchronized area as shown in the diagram.

Block level synchronization :-

In block level synchronization the entire method does not get synchronized, only the part of the method gets synchronized so all the thread will enter inside the method but only one thread will enter inside the synchronized block as shown in the diagram below.

Note :- In between method level synchronization and block level synchronization, block level synchronization is more preferable because all the threads can enter inside the method so only one part of the method is synchronized so only one thread will enter inside the synchronized block.

How synchronization machanism controls multiple threads :-

Every Object has a lock(monitor) in java environment and this lock can be given to only one Thread at a time.

The thread who acquires the lock will enter inside the synchronized area, it will complete its task without any disturbance because at a time there will be only one thread inside the synchronized area. This is known as Thread-safety in java.

The thread which is inside the synchronized area, after completion its task while going back will release the lock so the other threads (which are waiting outside for the lock) will get a chance to enter inside the synchronized area by again taking the lock from the Object and submitting it to the synchronization mechanism.

Note :-

---

synchronization logic can be written by senior programmer in IT industry because due to poor

synchronization, there may be chance of getting deadlock.

Program on method level synchronization :

```
class Table
{
	public synchronized void printTable(int num)
	{
		for (int i=1; i<=10; i++)
		{
			System.out.println(num*i);

			try
			{
				Thread.sleep(500);
			}
			catch(Exception e) {}
		}
	}
}

class Thread1 extends Thread
{
```

```
        Table t ;
        Thread1(Table t)  //t = obj
        {
                this.t = t;
        }

        public void run()
        {
                t.printTable(10);
        }
}

class Thread2 extends Thread
{
        Table t;
        Thread2(Table t)
        {
                this.t = t;
        }

        public void run()
        {
                t.printTable(20);
        }
```

```
}


public class MethodLevelSynchronization
{
	public static void main(String[] args)
	{
		Table obj = new Table();


		Thread1 t1 = new Thread1(obj);
		Thread2 t2 = new Thread2(obj);


		t1.start();
		t2.start();


	}


}
```

Problem with Object level synchronization :-

From the above diagram it is clear that there is no interference between t1 and t2 thread because they are passing throgh Object1 whereas on the other hand there is no interferenec even in between t3 and t4 because they are also passing through Object2.

But there may be chance that with t1 Thread, t3 or t4 can enter inside the synchronized area at the same time, simillarly it is also possible that with t2 thread, t3 or t4 can enter inside the

synchronized area so the conclusion is synchronization mechanism will not work with multiple Objects.

```
class Table1
{
   public synchronized void printTable(int n)
   {
     for(int i=1; i<=10; i++)
     {
            try
            {
                    Thread.sleep(100);
            }
            catch(InterruptedException e)
            {
                    System.err.println("Thread is Interrupted...");
            }
            System.out.println(n*i);
     }
   }
}
class Thread5 extends Thread
{
          Table1 t;
          Thread5(Table1 t)  //t = obj
```

```
        {
          this.t = t;
        }
        @Override
        public void run()
        {
                t.printTable(5);
        }
}
class Thread6 extends Thread
{
        Table1 t;
        Thread6(Table1 t)
        {
          this.t = t;
        }
        @Override
        public void run()
        {
                t.printTable(10);
        }
}
public class ProblemWithObjectSynchronization
{
```

```
        public static void main(String[] args)
        {
          Table1 obj1 = new Table1();     //object 1
          Thread5 t1 = new Thread5(obj1);
          Thread6 t2 = new Thread6(obj1);

          Table1 obj2 = new Table1();  //object 2
          Thread5 t3 = new Thread5(obj2);
          Thread6 t4 = new Thread6(obj2);

          t1.start();  t2.start(); t3.start(); t4.start();
        }
}
```

In order to solve Object level synchronization with multiple Objects, we introduced static synchronization.

static synchronization means now we will declare one static synchronization method so now the lock will be on the class but not object.

```
class Table
{
        public static synchronized void printTable(int num)
        {
                for (int i=1; i<=10; i++)
```

```
                {
                        System.out.println(num*i);

                        try
                        {
                                Thread.sleep(500);  // t1 and t2
                        }
                        catch(Exception e) {}
                }
        }
}

class MyThread1 extends Thread
{
        @Override
        public void run()
        {
                Table.printTable(5);
        }
}

class MyThread2 extends Thread
{
        @Override
```

```
	public void run()
	{
		Table.printTable(10);
	}
}

public class StaticSynchronization
{
	public static void main(String[] args)
	{
		MyThread1 t1 = new MyThread1();
		MyThread2 t2 = new MyThread2();

		t1.start();  t2.start();
	}
}
```

Thread priority :-

It is possible in java to assign priority to a Thread. Thread class has provided two predefined method setPriority(int priority) and getPriority() to set and get the priority of the thread respectively.

In java we can set the priority of the Thread in number from 1- 10 only where 1 is the minimum priority and 10 is the maximum priority.

Whenever we create a thread in java by default its priority would be 5 that is normal priority.

Thread class has also provided 3 final static variables which are as follows :-

Thread.MIN_PRIORITY :- 01

Thread.NORM_PRIORITY : 05

Thread.MAX_PRIORITY :- 10

Note :- We can't set the priority of the Thread beyond the limit so if we set the priority beyond the limit (1 to 10) then it will generate an exception java.lang.IllegalArgumentException

---

```
public class MainPriority
{
	public static void main(String[] args)
	{
		Thread t = Thread.currentThread();



		System.out.println("Main thread priority is :"+t.getPriority());


		Thread t1 = new Thread();
		System.out.println("User thread priority is :"+t1.getPriority());
```

```
        }


}
```

Note :- User defined thread is created as a part of main Thread hence it will acquire the priority from the main Thread only.

```
class ThreadP extends Thread
{
        @Override
        public void run()
        {
          int priority = Thread.currentThread().getPriority();
          System.out.println("Child Thread priority is :"+priority);
        }
}
public class MainPriority1
{
        public static void main(String[] args)
        {
                Thread t = Thread.currentThread();


                t.setPriority(Thread.MAX_PRIORITY);  //10


                //t.setPriority(11); -> java.lang.IllegalArgumentException
```

```
                System.out.println("Main thread priority is :"+t.getPriority());


                ThreadP t1 = new ThreadP();

                t1.start();

        }

}
```

Inter thread communication :-

It is a mechanism to communicate two synchronized threads within the context to acheive a particular task.

In ITC we put a thread into wait mode by using wait() method and other thread will complete its task, after completion it will call notify() method so the waiting thread will get a notification to complete its remaining task.

ITC can be implemented by the following method of Object class.

1) public final void wait() throws InterruptedException

2) public final void notify()

3) public final void notifyAll()

public final void wait() throws InterruptedException :-

It will put a thread into temporarly waiting state and it will release the lock.

It will wait till the another thread invokes notify() or notifyAll() for this object.

public final void notify() :-

It will wake up the single thread that is waiting on the same object.

public final void notifyAll() :-

It will wake up all the threads which are waiting on the object.

Note :- wait(), notify() and notifyAll() methods are defined in Object class but not in Thread class because methods are related to lock and Object has a lock so, all these methods are defined inside Object class.

Wap in java that describes the problem we will face if we dno't use inter thread communication :-

```
class Test implements Runnable
{
        int var=0;
```

```
        @Override
        public void run()
        {
                for(int i=2; i<=10; i++)
                {
                        var = var + i;
                        try
                        {
                                Thread.sleep(100);
                        }
                        catch (Exception e)
                        {
                        }
                }
        }
}
class ITCProblem
{
        public static void main(String[] args)
        {
                Test t = new Test();
                Thread t1 = new Thread(t) ;
                t1.start();
                try
```

```
                {
                        Thread.sleep(100);
                }
                catch (Exception e)
                {
                }
                System.out.println(t.var);
        }
}
```

```
class InterThreadComm
{
public static void main(String [] args)
{
SecondThread b = new SecondThread();
b.start();

                synchronized(b)
                        {
                                try
                                {
                                        System.out.println("Waiting for b to complete...");
                                        b.wait();   // after releasing the lock, waiting here
```

```
                                        }
                                        catch (InterruptedException e)
                                        {


                                        }
                                        System.out.println("Value is: " + b.value);
                                }
        }
}
class SecondThread extends Thread
{
        int value = 0;
        @Override
                public void run()
                {
                        synchronized(this)
                        {
                                for(int i=1;i<=5;i++)
                                {
                                        value =  value +i;
                                }
                                notify();  //will give notification to waiting thread
                        }
         }
```

```
}

---------------------------------------------------------------------------------

class Customer

{

int balance=10000;


        synchronized void withdraw(int amount)

        {

                System.out.println("going to withdraw...");

                if(balance<amount)

                        {

                                System.out.println("Less balance; waiting for deposit...");

                                        try

                                        {

                                                wait();

                                        }

                                        catch(Exception e){}

                        }

                balance  = balance - amount;

                System.out.println("withdraw completed..."+balance+" is remaining balance");

        }


        synchronized void deposit(int amount)
```

```
                {
                        System.out.println("going to deposit...");
                        balance = balance + amount;
                        System.out.println("deposit completed... ");
                        notify();
                }
}
class InterThreadBalance
{
public static void main(String args[])
        {
    Customer c=new Customer();

                new Thread()  //anonymous class concept
                {
                        public void run()
                        {
                                c.withdraw(15000);
                        }
                }.start();

                new Thread()
                {
                        public void run()
```

```
                {
                        c.deposit(10000);
                }
        }.start();
    }
}
```

22-FEB-22

Collection Framework :

Collection framework is nothing but handling group of Objects. We know only object can move from one network to another network.

A collection framework is a class library to handle group of Objects.

It is implemented by using java.util package.

It provides an architecture to store an manipulate group of objects.

All the operation that we can perform on data such as searching, sorting, insertion and deletion can be done by using collection framework because It is the data structure of Java.

The simple meaning of collection is single unit of Objects.

It provides the following interfaces :

1) List (Accept duplicate elements)

2) Set (Not accepting duplicate elements)

3) Queue (Storing and Fetching the elements based on some order)

The following are the methods of Collection(I) interface:-

---

a) public boolean add(Object element) :- It is used to add an item/element in the collection.

b) public boolean addAll(Collection c) :- It is used to insert the specified collection elements in the existing collection

c) public boolean remove(Object element) :- It is used to delete an element from the collection.

d) public boolean removeAll(Collection c) :- It is used to delete all the elements from the existing collection.

List :

It is predefined interface, which is the sub interface of collection.

It contains List of elements having duplicate values.

It stores the element on the basis of index.

It can perform sorting Operation

It can hetrogeneous types of data

Behavior of List interface Collection classes :

---

1) Unlike array It accepts hetrogeneous type of data (different kinds of data)

2) Just like array List interface collection classes store the element on the

basis of index.

3) It is dynamically growable in nature but we have some performance issue to copy paste the old data to the new memory.

So if we know the size in advance it is better to go with Array but if we don't know the size then go with collection.

```
import java.util.*;
class CollectionDemo
{
	public static void main(String[] args)
	{
			Object []x = new String[3];
```

```
                x[0]= "Raj";
                x[1] = "Naresh";
                x[2] = 13;              //ArrayStoreException
        }
}
```

How to iterarte the elements from the collection :

Iterator interface :-

It provides the facility to read the data from the collection in forward direction only.

It has two important method

1) public boolean hasNext() :- The data is available in the next position or not, if available it will return true otherwise it will return false.

2) public Object next() :- It will read the data from the collection that is the reason the return type is object.

ListIterator interface :-

It is used to read the data in both the directions i.e forward as well as backward direction.

1) public boolean hasNext()

2) public Object next()

3) public boolean hasPrevious()

4) public Object previous()

ArrayList :-

ArrayList :-

It is a predefined class available in java.util package under List interface.

It accepts duplicate elemnts and null values.

It is dynamically growable array.

It stores the elements on index basis so it is simillar to an array.

Initial capacity of ArrayList is 10.The capacity of Arraylist can be calculated by using the formula

new capacity = (old capacity * 3/2) + 1

All the methods declared inside an ArrayList is not synchronized so multiple thread can access the method of ArrayList.

It implements List,Serializable, Clonable, RandomAccess interfcaes

---

Note :- Collections is a predefined class in java.util where as Collection is an interface in java.util.

```
import java.util.*;

class ArrayListDemo

{

        public static void main(String... a)

        {

                ArrayList<String> arl = new ArrayList<String>();


                arl.add("Apple");

                arl.add("Orange");

                arl.add("Grapes");

                arl.add("Mango");

                arl.add("Guava");

                arl.add("Mango");


                System.out.println("Contents :"+arl);


                arl.remove(2);

                arl.remove("Guava");
```

```
                System.out.println("Contents After Removing :"+arl);
                System.out.println("Size of the ArrayList:"+arl.size());

                Collections.sort(arl);

                for(String x : arl)
                {
        System.out.println(x);
                }
           }
}
```

```
import java.util.*;
class Employee
 {
        int eno;
        String name;
        int age;

        Employee(int eno,String name,int age)
                 {
                        this.eno=eno;
                        this.name=name;
                        this.age=age;
```

```
                    }
 }

class ArrayListDemo1
{
  public static void main(String args[])
          {
            Employee e1 = new Employee(111,"Raj",23);
            Employee e2 = new Employee(222,"Aryan",24);
            Employee e3 = new Employee(333,"Puja",25);

            ArrayList<Employee> al = new ArrayList<Employee>();
            al.add(e1);
            al.add(e2);
            al.add(e3);

            Iterator itr = al.iterator();

            while(itr.hasNext())
                    {
                       Employee obj =(Employee) itr.next();
                              System.out.println("Employee id is :"+obj.eno);
                              System.out.println("Employee name is :"+obj.name);
                              System.out.println("Employee age is :"+obj.age);
```

```
                        System.out.println(".....");
                }
  }
}
```

```
//Program to merge two collection
import java.util.*;
class ArrayListDemo2
        {
                public static void main(String args[])
                {
                 ArrayList<String> al1=new ArrayList<String>();
                 al1.add("Ravi");
                 al1.add("Rahul");
                 al1.add("Rohit");


                 ArrayList<String> al2=new ArrayList<String>();
                 al2.add("Pallavi");
                 al2.add("Sweta");

                 al1.addAll(al2);

                 Iterator itr=al1.iterator();
```

```
                    while(itr.hasNext())
                    {
                     System.out.println(itr.next());
                    }
    }
}
```

//Program to fetch the elements in forward and backward direction using ListIterator interface

```
import java.util.*;
class ArrayListDemo3
{
  public static void main(String args[])
        {
                ArrayList<String> al=new ArrayList<String>();
                al.add("Pallavi");
                al.add("Ravi");
                al.add("Rahul");
                al.add("Sachin");
                al.add("Aswin");
                al.add("Ananya");
                al.add("Bina");
```

```
                System.out.println("element at 2nd position: "+al.get(2));
                Collections.sort(al);
                Collections.reverse(al);

                ListIterator itr=al.listIterator();

                System.out.println("traversing elements in forward direction...");
                while(itr.hasNext())
                {
                        System.out.println(itr.next());
                }

                System.out.println("traversing elements in backward direction...");
                while(itr.hasPrevious())
                {
                System.out.println(itr.previous());
                }
        }
}
```

```
////Serialization
import java.io.*;
import java.util.*;
 class ArrayListDemo4
```

```
    {
public static void main(String [] args)
{
  ArrayList<String> al=new ArrayList<String>();
  al.add("Nagpur");
  al.add("Vijaywada");
  al.add("Hyderabad");
                al.add("Jamshedpur");
 try
 {
                                //Serialization
                                FileOutputStream fos=new FileOutputStream("City.txt");
                                ObjectOutputStream oos=new ObjectOutputStream(fos);
                                 oos.writeObject(al);
                                 fos.close();
                                 oos.close();
                                //Deserialization
                                FileInputStream fis=new FileInputStream("City.txt");
                                ObjectInputStream ois=new ObjectInputStream(fis);

                                ArrayList  list=(ArrayList)ois.readObject();
                                System.out.println(list);
 }
              catch(Exception e)
```

```
        {
           System.err.println(e);
        }
      }
   }
```

```
import java.util.ArrayList;


class ArrayListDemo5
{
           public static void main(String[] args)
           {
                      ArrayList<String> city= new ArrayList<String>();//default capacity is 10
                      city.ensureCapacity(3);//resized the arraylist to store 3 elements.

                      city.add("Hyderabad");
                      city.add("Mumbai");
                      city.add("Delhi");

                      city.add("Kolkata");
                      System.out.println("ArrayList: " + city);
            }
}
```

//Program on ArrayList that contains null values as well as we can pass the element position basis

```
import java.util.*;

class ArrayListDemo6

{

	public static void main(String[] args)

	{

		ArrayList al = new ArrayList(); //raw type(unsafe operation)

		al.add(12);

		al.add("Ravi");

		al.add(12);

		al.add(0,"Hyderabad"); //add(int index, Object o)method of List interface

		al.add(1,"Naresh");

		al.add(null);

		al.add(11);

		System.out.println(al);

	}

}
```

LinkedList :-

It is a predefined class available in java.util package under List interface.

It is ordered by index position like ArrayList except the elements (nodes) are doubly linked to one another. This linkage gives us new method for adding and removing the elements from the middle of LinkedList.

*The important thing is, LikedList may iterate more slowly than ArrayList but LinkedList is a good choice when we want to insert or delete the elements frequently in the list.

From jdk 1.5 onwards LinkedList has enhanced to support basic queue operation.

Constructor:

It has 2 constructors

1) LinkedList list1 = new LinkedList();

   It will create a LinkedList object with 0 capacity.

2) LinkedList list2 = new LinkedList(Collection c);

   Interconversion between the collection

Methods of LinkedList class:

---

1) void addFirst(Object o)

2) void addLast(Object o)

3) Object getFirst()

4) Object getLast()

5) Object removeFirst()

6) Object removeLast()

```
import java.util.*;
class LinkedListDemo
{
 public static void main(String args[])
 {
     List list=new LinkedList();
              list.add("Ravi");
              list.add("Vijay");
              list.add("Ravi");
              list.add(null);
              list.add(42);
              Iterator itr=list.iterator();
              while(itr.hasNext())
              {
               System.out.println(itr.next());
              }
  }
}
```

```
import java.util.*;
class LinkedListDemo1
{
    public static void main(String args[])
    {
        LinkedList<String> list= new LinkedList<String>();
        list.add("Item 2");//2
        list.add("Item 3");//3
        list.add("Item 4");//4
        list.add("Item 5");//5
        list.add("Item 6");//6
        list.add("Item 7");//7
        list.add("Item 9"); //10

        list.add(0,"Item 0");//0
        list.add(1,"Item 1"); //1
        list.add(8,"Item 8");//8
                    list.add(9,"Item 10");//9
         System.out.println(list);

         list.remove("Item 5");
         System.out.println(list);
```

```
        list.removeLast();
        System.out.println(list);

                        list.removeFirst();
        System.out.println(list);

       list.set(0,"Ajay"); //set() will override the existing value
       list.set(1,"vijay");
       list.set(2,"Anand");
       list.set(3,"Aman");
       list.set(4,"Suresh");
       list.set(5,"Ganesh");
       list.set(6,"Harsh");
        System.out.println(list);

        for (String str : list)
        {
                                System.out.println(str);
        }
    }
}
```

```
import java.util.LinkedList;
public class LinkedListDemo2
```

```
{
    public static void main(String[] argv)
    {
        LinkedList list = new LinkedList();
        list.addFirst("Ravi");
        list.add("Rahul");
        list.addLast("Anand");
        System.out.println(list.getFirst());
        System.out.println(list.getLast());
        list.removeFirst();
        list.removeLast();
        System.out.println(list); //[Rahul]
    }
}
```

ListIterator interface has provided some more methods apart from reading the elements from forward sirection and backward direction.

1) public boolean hasNext()

2) public Object next()

3) public boolean hasPrevious()

4) public Object previous()

5) public void remove()

6) public void add(Object newElement)

7) public void set(Object newElement)

```
import java.util.*;
class LinkedListDemo3
{
        public static void main(String[] args)
        {
                LinkedList city = new LinkedList();
    city.add("Kolkata");
                city.add("Bangalore");
                city.add("Hyderabad");
                city.add("Pune");
                System.out.println(city);

                ListIterator lt = city.listIterator();

    while(lt.hasNext())
                {
```

```
                String x = (String) lt.next();


                if(x.equals("Kolkata"))
                {
        lt.remove();
                }


                else if(x.equals("Hyderabad"))
                {
        lt.add("Ameerpet");
                }


                else if(x.equals("Pune"))
                {
        lt.set("Mumbai");
                }
            }


            for(Object o : city)
            {
                System.out.println(o);
            }
    }
}
```

Vector :-

Vector is a predefined class available in java.util package under List interface. Vector is always from java means it is available from jdk 1.0 version.

Vector and Hashtable, these two classes are available from jdk 1.0, remaining classes were added from 1.2 version. That is the reason Vector and Hashtable are called legacy classes.

The main difference between Vector and ArrayList is ArrayList methods are not synchronized so multiple threads can access the method of ArrayList where as on the other hand most the methods are synchronized in Vector so performance wise Vector is slow.

*We should go with ArrayList when Threadsafety is not required on the other hand we should go with Vector when we need ThreadSafety

It also stores the elements on index basis.It is dynamically growable with initial capacity 10.

Just like ArrayList it also implments List, Serializable, Clonable, RandomAccess interfaces.

Ex:-

public class Vector extends AbstractList implements List, Serializable, Clonable, RandomAccess

Constructor in Vector :

---

1) Vector v1 = new Vector();

It will create the vector object with default capacity is 10

new capacity = old capacity * 2 => new capacity = 10 * 2 = 20

2) Vector v2 = new Vector(int initialCapacity);

Will create the vector object with specified capacity.

3) Vector v3 = new Vector(int initialCapacity, int incrementalCapacity);

Eg :- new Vector(1000,5);

Initialy It will create the Vector Object with initial capacity 1000 and then when when the capacity will full then increment by 5 so the nect capacity would be 1005, 1010 and so on.

4) Vector v4 = new Vector(Collection c);

```
import java.util.*;
class VectorExampleDemo1
{
        public static void main(String[] args)
        {
```

```
                Vector v = new Vector();
        v.add(12);
                 v.add("Ravi");
                 v.add(3);
                 v.add(4);
                 v.add(null);
                System.out.println(v);  //v.toString(); [12,Ravi,3,4,null]
                System.out.println(".................");
                System.out.println("It's capacity is :"+v.capacity());
           }
}
```

```
 import java.util.Vector;
 class VectorExampleDemo2
           {
             public static void main(String[] args)
                      {
                               Vector v = new Vector();
                               v.add(1);
                               v.add("2");
                               v.add(34.90f);
                               System.out.println(v);
                               System.out.println("Getting elements of Vector");
                               System.out.println(v.get(0));
```

```
                    System.out.println(v.get(1));
                    System.out.println(v.get(2));
            }
}
```

```
import java.util.*;
class VectorExampleDemo3
{
        public static void main(String args[])
        {
                Vector<Integer> v = new Vector<Integer>();

                int x[]={22,20,10,40,15,58};

                for(int i=0; i<x.length; i++)
                {
                        v.add(x[i]);
                }
                Collections.sort(v);
                System.out.println("Maximum element is :"+Collections.max(v));
                System.out.println("Minimum element is :"+Collections.min(v));
                System.out.println("Vector Elements :");

                for(int i=0; i<v.size();i++)
```

```
                {
                System.out.println(v.get(i));
                }
        }
}
```

Stack :

It is a predefined class available in java.util package. It is the sub class of Vector class.

It is a linear data structure that is used to store the Objects in LIFO (Last In first out) basis.

Inserting an element into a Stack is known as push operation and It can be done by push() method whereas extracting an element from the stack is known as pop operation and it can be done by pop(). pop() method will extract and delete the element from the top of the stack.

If we want to extract the element from the top of the Stack without removing it then we should use peek() of Stack class.

It throws an exception called EmptyStackException.

It has only one constructor as shown below

```
Stack s = new Stack();
```

Method :

---

push(Object o) :- To insert an element

pop() :- To remove and return the element from the top of the Stack

peek() :- Will fetch the element from top of the Stack without removing

empty() :- Tests whether stack is empty or not (boolean return type)

search(Object o) :- It will search a particular element in the Stack and it returns OffSet. If the element is not present in the Stack it will return -1

```
import java.util.*;
class Stack1
{
    public static void main(String args[])
    {
        Stack<Integer> s = new Stack<Integer>();
        try
        {
            s.push(0);
            s.push(1);
                                    s.push(2);
```

```
                                        s.push(3);
                                        s.push(4);
                                        s.push(5);
                                        System.out.println(s.pop());
                                        System.out.println(s.pop());
                                        System.out.println(s.pop());
                                        System.out.println(s.pop());
                                        System.out.println(s.pop());
                                        System.out.println(s.pop());
                                        System.out.println(s);
            }
                                catch(EmptyStackException e)
                                {}
    }
}
```

```
//add() is the method of Vector class
import java.util.*;
class Stack2
{
    public static void main(String args[])
    {
        Stack<Integer> st1 = new Stack<Integer>();
        st1.add(10);
```

```
st1.add(20);
for(int k : st1)
{
    System.out.println(k+"   ");
}
Stack<String> st2 = new Stack<String>();
st2.add("Java");
st2.add("is");
st2.add("programming");
st2.add("language");
for(String k : st2)
{
    System.out.println(k+"   ");
}
Stack<Character> st3 = new Stack<Character>();
st3.add('A');
st3.add('B');
for(Character k : st3)
{
    System.out.println(k+"   ");
}
Stack<Double> st4 = new Stack<Double>();
st4.add(10.5);
st4.add(20.5);
```

```
        for(Double k : st4)
        {
            System.out.println(k+"   ");
        }
    }
}
```

```
import java.util.Stack;
public class Stack3
{
          public static void main(String[] args)
                    {
                              Stack<String> stk= new Stack<String>();
                              stk.push("Apple");
                              stk.push("Grapes");
                              stk.push("Mango");
                              stk.push("Orange");
                              System.out.println("Stack: " + stk);

                              String fruit = stk.peek();
                              System.out.println("Element at top: " + fruit);
                              System.out.println("Stack: " + stk);
                    }
}
```

Set interfcae :

Set interface doesnot accept duplicate elements here our friend is equals(Object) method of Object class which compares two objects and if both objects are identical it will accept only one.

Set interface does not provide any own method, it inherits all the methods from Collection interface.

26-Feb-22

HashSet :-

---

public class HashSet extends AbstractSet implements Set, Clonabale, Serializable

It is a predefined class available in java.util package under Set interface.

It is an unsorted and unordered set.

It accepts hetrogeneous kind of data.

It uses the hashcode of the object being inserted into the Collection.

It doesn't contain any duplicate elements as well as It does not maintain any order while

iterating the elements from the collection.

It can accept null value.

HashSet is used for searching operation.

```
import java.util.*;
class HashSetDemo
{
 public static void main(String args[])
 {
            HashSet<String> hs=new HashSet<String>();
            hs.add("Ravi");
            hs.add("Vijay");
            hs.add("Ravi");
            hs.add("Ajay");
            hs.add("Palavi");
            hs.add("Sweta");
            hs.add(null);
            hs.add(null);

   // Collections.sort(hs);          //Invalid not possible

            Iterator itr=hs.iterator();
```

```
            while(itr.hasNext())
            {
             System.out.println(itr.next());
            }
  }
}
```

```
import java.util.*;
public class HashSetDemo1
{
    public static void main(String[] argv)
    {
        boolean[] ba = new boolean[6];

        Set s = new HashSet();

        ba[0] = s.add("a");
        ba[1] = s.add(42);
        ba[2] = s.add("b");
        ba[3] = s.add("a");
        ba[4] = s.add("new Object()");
                        ba[5] = s.add(new Object());

        for(int x = 0; x<ba.length; x++)
```

```
            System.out.print(ba[x]+"    ");


        System.out.println("\n");


                            for(Object o : s)

            System.out.print(o+"   ");

    }

}
```

LinkedHashSet :-

It is a predefined class in java.util package under Set interface.

It is the sub class of HashSet class

It is an orderd version of HashSet that maintains a doubly linked list across all the elements.

We should use LinkedHashSet class when we want to maintain an order.

When we iterate the elements through HashSet the order will be unpredictable, while when we iterate the elments through LinkedHashSet then the order will be same as they were inserted in the collection.

It accepts hetrogeneous and null value is allowed.

```
import java.util.*;
class LinkedHashSetDemo
{
 public static void main(String args[])
        {
                LinkedHashSet<String> al=new LinkedHashSet<String>();
                al.add("Ravi");
                al.add("Vijay");
                al.add("Ravi");
                al.add("Ajay");
                al.add("Pawan");

               //Collections.sort(al);  //not possible

                Iterator itr=al.iterator();
                while(itr.hasNext())
                {
                 System.out.println(itr.next());
                }
        }
}
```

SortedSet :

---

1) It is the sub interface of Set interface

2) If we don't want duplicate elements and wants to store the elements based on some sorting order i.e default natural sorting order then we should go with SortedSet(I)

3) We have two interfaces Comparable(available in java.lang package) and Comparator (java.util package) to compare two objects.

What is the difference between Comparable and Comparator interface :

```
public class Customer implements Comparable<Customer>
{
  int cid;
  String cname;
  int cage;

  public Customer(int cid, String cname, int cage)
  {
        super();
        this.cid = cid;
        this.cname = cname;
```

```
        this.cage = cage;
    }

        @Override
        public int compareTo(Customer ct)
        {
                return this.cname.compareTo(ct.cname);

        }

}

import java.util.ArrayList;
import java.util.Collections;

public class CustomerComparable
{
        public static void main(String[] args)
        {
                ArrayList<Customer> al = new ArrayList<Customer>();
                al.add(new Customer(111,"Rahul",24));
                al.add(new Customer(333,"Aryan",27));
                al.add(new Customer(222,"Zaheer",30));
```

```
                Collections.sort(al);


                for(Object obj : al)
                {
                        Customer c = (Customer) obj;
                        System.out.println("customer cid = "+c.cid+" Customer name ="+c.cname+" Customer age ="+c.cage);
                }


        }
}
```

Program on Comparator interface :

```
//BLC
public class Employee
{
  int eid;
  String ename;
  String eaddr;

        public Employee(int eid, String ename, String eaddr)
```

```
	{
		super();
		this.eid = eid;
		this.ename = ename;
		this.eaddr = eaddr;
	}
}

//ELC

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class EmployeeComparator
{
	public static void main(String[] args)
	{
	  ArrayList<Employee> al = new ArrayList<Employee>();
	  al.add(new Employee(101,"Raj","Hyderabad"));
	  al.add(new Employee(103,"Zaheer","Amrawati"));
	  al.add(new Employee(102,"Aryan"," S R Narag"));

	  //sorting on the basis of Employee Id
```

```
System.out.println("Sorted based on Employee Id");
Comparator<Employee> cmpId = new Comparator<Employee>()
{
	@Override
	public int compare(Employee e1, Employee e2)
	{
		return e1.eid - e2.eid;
	}
};

Collections.sort(al, cmpId);

for(Employee e : al)
{
	 System.out.println(e.eid+" : "+e.ename+" : "+e.eaddr);
}


//Sorting on the basis of Employee name
System.out.println("-----------------------------------");
System.out.println("Sorted based on Employee Name");
Comparator<Employee> cmpName = new Comparator<Employee>()
{
	@Override
```

```
                public int compare(Employee e1, Employee e2)
                {
                        return e1.ename.compareTo(e2.ename);
                }
            };

            Collections.sort(al, cmpName);

            for(Employee e : al)
            {
                System.out.println(e.eid+" : "+e.ename+" : "+e.eaddr);
            }

          }
}
```

TreeSet :

It is a predefined class available in java.util package under Set interface.

TreeSet and TreeMap are the two sorted collection in the entire Collection Framework so both the classes never accepting hetrogeneous kind of the data.

It will sort the elements in natural sorting order i.e ascending order in case of number and

alphabetical order in the case of String. In order to sort the elements It uses Comparable interface.

It does not accept duplicate and null value.

It does not accept hetrogeneous type of data if we try to insert it will throw a runtime exception i.e java.lang.ClassCastException

TreeSet implements NavigableSet.

NavigableSet extends SortedSet

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String[] args)
	{
		Set t = new TreeSet();
		t.add(4);
		t.add(7);
		t.add(2);
		t.add(1);
		t.add(9);
		t.add("Raj"); //we cannot take hetrogeneous
```

```
                System.out.println(t);
        }
}
```

```
import java.util.*;
class TreeSetDemo
{
        public static void main(String[] args)
        {
                Set t = new TreeSet();
                t.add(4);
                t.add(7);
                t.add(2);
                t.add(1);
                t.add(9);

                System.out.println(t);
        }
}
//Output of this program is naturally sorted i.e [1,2,4,7,9]
```

```
import java.util.*;
class TreeSetDemo1
{
	public static void main(String[] args)
	{
		Set<String> t = new TreeSet<String>();
		t.add("Orange");
		t.add("Mango");
		t.add("Pear");
		t.add("Banana");
		t.add("Apple");
		System.out.println(t);
	}
}
```

```
import java.util.*;
class TreeSetDemo2
{
	public static void main(String[] args)
	{
		Set<String> t = new TreeSet<String>();
		t.add("6");
		t.add("5");
		t.add("4");
```

```
                t.add("2");
                t.add("9");
                System.out.println(t);
        }
}
/Output of this program is naturally sorted i.e [2,4,5,6,9]
---------------------------------------------------------------------------
import java.util.*;
public class TreeSetMethodDemo
{
    public static void main(String[] args)
    {
        TreeSet<Integer> times = new TreeSet<Integer>();
        times.add(1205);
        times.add(1505);
        times.add(1545);
            times.add(1600);
        times.add(1830);
        times.add(2010);
        times.add(2100);

        TreeSet<Integer> sub = new TreeSet<Integer>();

                        sub =  (TreeSet<Integer>) times.subSet(1545,2100);
```

```
        System.out.println("Using subSet() :-"+sub);//[1545...2010]
        System.out.println(sub.first());
        System.out.println(sub.last());

                        sub =  (TreeSet<Integer>)times.headSet(1545);
                        System.out.println("Using headSet() :-"+sub);

                        sub =  (TreeSet<Integer>)times.tailSet(1545);
                        System.out.println("Using tailSet() :-"+sub);
    }
}
```

Map interface :

Why Map interface is not the part of Collection(I)?

Collection interface deals with individual Object but Map interface deals with group of Objects in the form of {key=value} pair.

HashMap :

It is a predefined class available in java.util package under Map interface.

It gives us unsorted and Unordered map. when we need a map and we don't care about the order while iterating the elements through it then we should use HashMap.

It is unsynchronized and provides a gurantee that the order will remain constant.

It inserts the element based on the hashCode of the Object key using hashing technique.

It does not accept duplicate keys but value may be duplicate.

It accepts only one null key(because duplicate keys are not allowed) but multiple null values.

HashMap is not synchronized.

Time complexcity of search, insert and delete will be O(1)

We should use HashMap to perform searching opeartion.

```
import java.util.*;
public class HashMapDemo
{
    public static void main(String[] a)
    {
        Map<String,String> map = new HashMap<String,String>();
        map.put("Ravi", "12345");
                    map.put("Rahul", "12345");
                    map.put("Aswin", "5678");
```

```
                    map.put(null, "6390");
                    map.put("Ravi","1529");
        System.out.println(map.get(null));
                    System.out.println(map.get("virat"));
                    System.out.println(map);

    }
}
```

```
import java.util.*;

public class HashMapDemo1
{
        public static void main(String args[])
        {
                HashMap hm = new HashMap();
                hm.put(1, "JSE");
                hm.put(2, "JEE");
                hm.put(3, "JME");
                hm.put(4,"JavaFX");
                hm.put(5,null);

                System.out.println("Initial map elements: " + hm);
                System.out.println("key 2 is present or not :"+hm.containsKey(2));
```

```
                System.out.println("JME is present or not :"+hm.containsValue("JME"));
                System.out.println("Size of Map : " + hm.size());
                hm.clear();
                System.out.println("Map elements after clear: " + hm);
            }
}
```

```
//Collection view method by entrySet()
import java.util.*;
public class HashMapDemo2
{
public static void main(String args[])
        {
                        Map map = new HashMap();
                        map.put(1, "C");
                        map.put(2, "C++");
                        map.put(3, "Java");
                        map.put(4, ".net");
                        System.out.println(map);
                        System.out.println("Return old value :"+map.put(4,"Python"));

                        Set set=map.entrySet();
                        System.out.println("Set values: " + set); //[]
```

```
                Iterator<Map.Entry> itr = set.iterator();
                while(itr.hasNext())
            {
                        Map.Entry m =  itr.next();
                        System.out.println(m.getKey()+":"+m.getValue());
                        if(m.getKey().equals(4))
                    {
                            m.setValue("Advanced Java");
                    }
            }
                System.out.println(map);
        }
}
```

```
import java.util.*;

public class HashMapDemo3
{
public static void main(String args[])
 {
            HashMap<Integer,String> map = new HashMap<Integer,String>(10);
            map.put(1, "Java");
            map.put(2, "is");
```

```
                map.put(3, "best");
                map.remove(3);
                String val=(String)map.get(3);
                System.out.println("Value for key 3 is: " + val);
  }
}
```

```
import java.util.*;

public class HashMapDemo4
{
public static void main(String args[])
        {
                HashMap newmap1 = new HashMap();
                HashMap newmap2 = new HashMap();

                newmap1.put(1, "SCJP");
                newmap1.put(2, "is");
                newmap1.put(3, "best");

                System.out.println("Values in newmap1: "+ newmap1);

                newmap2.put(4, "Exam");
                newmap2.putAll(newmap1);
```

```
                    System.out.println("Values in newmap2: "+ newmap2);
  }
}
```

```
import java.util.*;
public class HashMapDemo5
{
    public static void main(String[] argv)
    {
        Map map = new HashMap(9, 0.85f);
        map.put("key", "value");
        map.put("key2", "value2");
        map.put("key3", "value3");

                    Set k_set = map.keySet();//keySet return type is Set
                    System.out.println(k_set );

        Collection v_set = map.values(); //values return type is collection
        System.out.println(v_set);

                    map.clear();
        System.out.println(map);
    }
```

```
}
--------------------------------------------------------------------------------
import java.util.*;
class  HashMapDemo6
{
	public static void main(String[] args)
	{
		Map<String, String> map = new HashMap<String, String>();

		map.put("A", "1");
		map.put("B", "2");
		map.put("C", "3");

		String value = map.getOrDefault("D", "Not Present");
		System.out.println(value);
	}
}
--------------------------------------------------------------------------------
//interconversion of two HashMap
import java.io.*;
import java.util.*;

class HashMapDemo7
	{
```

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

		Map<Integer, String> hm1 = new HashMap<Integer, String>();

		hm1.put(1, "Ravi");
		hm1.put(2, "Rahul");
		hm1.put(3, "Rajen");

	HashMap<Integer, String> hm2  = new HashMap<Integer,String>(hm1);

		System.out.println("Mappings of HashMap hm1 are : "   + hm1);

		System.out.println("Mapping of HashMap hm2 are : " + hm2);
	}
}
```

```
package com.ravi.hashmap;

import java.util.Objects;

public class Student
{
	int sid;
```

```
public Student(int sid)
{
	this.sid = sid;

}

@Override
public int hashCode() {
	return Objects.hash(sid);
}

@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	Student other = (Student) obj;
	return sid == other.sid;
}
```

```
}
```

```
import java.util.HashMap;

public class HashMapDemo
{
	public static void main(String[] args)
	{
		HashMap<Student,String> hm = new HashMap<Student,String>();
		hm.put(new Student(1),"Raj");
		hm.put(new Student(1),"Raj");
		System.out.println(hm.size());

		HashMap<Integer,String> hm1 = new HashMap<Integer,String>();
		hm1.put(1,"Raj");
		hm1.put(1,"Raj");
		System.out.println(hm1.size());


	}
}
```

Hashtable :

It is predefined class available in java.util package under Map interface.

Like Vector, Hashtable is also form the birth of java so called legacy class.

It is the sub class of Dictionary class which is an abstract class.

The major difference between HashMap and Hashtable is, HashMap can have null values as well as null keys where as Hashtable does not contain anything as a null. if we try to add null value JVM will throw an exception i.e NullPointerException.

The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Like Vector Hashtable methods are also synchronized So it provides ThreadSafety

It has also same constructor as we have in HashMap.(4 constructors)

```
import java.util.*;
class HashtableDemo
        {
         public static void main(String args[])
                {
                  Hashtable<Integer,String> map=new Hashtable<Integer,String>();
                  map.put(1, "Java");
```

```
                map.put(2, "is");
                map.put(3, "best");
                map.put(4,"language");
                System.out.println(map);

                for(Map.Entry m : map.entrySet())
                        {
                                System.out.println(m.getKey()+" = "+m.getValue());
                        }
        }
}
```

```
import java.util.*;
class HashtableDemo1
{
  public static void main(String args[])
          {
   Hashtable<Integer,String> map=new Hashtable<Integer,String>();
    map.put(1,"Priyanka");
    map.put(2,"Ruby");
    map.put(3,"Vibha");
    map.put(4,"Kanchan");
          //map.put(null,"Aarti");
          System.out.println("Initial Map: "+map);
```

```
    map.putIfAbsent(5,"Bina");
    System.out.println("Updated Map: "+map);

    map.putIfAbsent(1,"Priyanka");
    System.out.println("Updated Map: "+map);
 }
}
```

LinkedHashMap :

---

public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map

It is a predefined class available in java.util package under Map interface.

It is the sub class of HashMap class.

It maintains insertion order. It contains a doubly linked with the elements or nodes so It will iterate more slowly in comparison to HashMap.

It uses Hashtable and LinkedList data structure.

If We want to fetch the elements in the same order as they were inserted then we should go with LinkedHashMap.

It accepts one null key and multiple null values.

It is not synchronized.

It has also 4 constructors same as HashMap

1) LinkedHashMap hm1 = new LinkedHashMap();

   will create an empty LinkedHashMap

2) LinkedHashMap hm1 = new LinkedHashMap(iny initialCapacity);

3) LinkedHashMap hm1 = new LinkedHashMap(iny initialCapacity, float loadFactor);

4) LinkedHashMap hm1 = new LinkedHashMap(Map m);

```
import java.util.*;
class LinkedHashMapDemo
{
        public static void main(String[] args)
        {
                LinkedHashMap<Integer,String> l = new LinkedHashMap();
```

```
                    l.put(1,"abc");
                    l.put(3,"xyz");
                    l.put(2,"pqr");
                    l.put(4,"def");
                    l.put(null,"ghi");
                    System.out.println(l);
          }
}
```

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapDemo1
{
    public static void main(String[] a)
    {
        Map<String,String> map = new LinkedHashMap<String,String>();
        map.put("Ravi", "1234");
                    map.put("Rahul", "1234");
                    map.put("Aswin", "1456");
                    map.put("Samir", "1239");
        System.out.println(map);
    }
}
```

TreeMap :

---

public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V> , Clonable, Serializable

It is a predefined class avaialble in java.util package under Map interface.

It is a sorted map that means it will sort the elements by natural sorting order based on the key.

It will accept homogeneous kyes only.

It sorts the elements based on the key. In order to sort the key, it uses Comparable and Comparator interfaces.

It does not accept null key but null value we can take.

TreeMap implements NavigableMap and NavigableMap extends SortedMap.

TreeMap contains 4 types of Constructors :

1) TreeMap tm = new TreeMap(); //creates an empty TreeMap

2) TreeMap tm = new TreeMap(Comparator cmp); //user defined soting logic

3) TreeMap tm = new TreeMap(Map m);

4) TreeMap tm = new TreeMap(SortedMap m);

```
import java.util.*;
class TreeMapDemo
{
	public static void main(String[] args)
	{
		TreeMap t = new TreeMap();
		t.put(4,"Ravi");
		t.put(7,"Aswin");
		t.put(2,"Ananya");
		t.put(1,"Dinesh");
		t.put(9,"Ravi");
		//t.put(null,"Ashis");
		System.out.println(t);
	}
}
```

```
import java.util.*;
public class TreeMapDemo1
{
```

```
    public static void main(String args[])
    {
        TreeMap map = new TreeMap();
        map.put("one","1");
        map.put("two",null);
        map.put("three","3");
        displayMap(map);
    }
    static void displayMap(TreeMap map)
    {
        Collection c = map.entrySet();
        Iterator i = c.iterator();
        while(i.hasNext())
        {
            Object o = i.next();
            System.out.print(o + " ");
        }
    }
}
```

```
import java.util.*;
public class TreeMapDemo2
{
```

```
   public static void main(String[] argv)
   {
      Map map = new TreeMap();
      map.put("key2", "value2");
      map.put("key3", "value3");
      map.put("key1", "value1");

      System.out.println(map);
      SortedMap x = (SortedMap) map;
      System.out.println("First key is :"+x.firstKey());
      System.out.println("Last Key is :"+x.lastKey());
    }
}
```

```
//customized sorting order
import java.io.*;
import java.util.*;

class TreeMapDemo3
        {
        public static void main(String[] args)
        {
                System.out.println("Sorting name -> Ascending Order");

                TreeMap<Employee,String>  tm1  =  new  TreeMap<Employee,String>(new
```

```
FirstComparator());

        tm1.put(new Employee(101, "Zaheer", 24),"Hyderabad");
        tm1.put(new Employee(201, "Aryan", 27),"Jamshedpur");
        tm1.put(new Employee(301, "Pooja", 26),"Mumbai");

        System.out.println(tm1);

    System.out.println("----------------------------------------");

        System.out.println("Sorting name -> Descending Order");

        TreeMap<Employee,String> tm2 = new TreeMap<Employee,String>(new
SecondComparator());

        tm2.put(new Employee(101, "Zaheer", 24),"Hyderabad");
        tm2.put(new Employee(201, "Aryan", 27),"Jamshedpur");
        tm2.put(new Employee(301, "Pooja", 26),"Mumbai");

        System.out.println(tm2);

  System.out.println("----------------------------------------");
        System.out.println("Sorting Age -> Ascending Order");
        TreeMap<Employee,String> tm3 = new TreeMap<Employee,String>(new
ThirdComparator());
```

```
                tm3.put(new Employee(101, "Zaheer", 24),"Hyderabad");
                tm3.put(new Employee(201, "Aryan", 27),"Jamshedpur");
                tm3.put(new Employee(301, "Pooja", 26),"Mumbai");

                System.out.println(tm3);

                System.out.println("----------------------------------------");
                System.out.println("Sorting Age ->  Descending Order");
                TreeMap<Employee,String>  tm4  =  new  TreeMap<Employee,String>(new
FourthComparator());

                tm4.put(new Employee(101, "Zaheer", 24),"Hyderabad");
                tm4.put(new Employee(201, "Aryan", 27),"Jamshedpur");
                tm4.put(new Employee(301, "Pooja", 26),"Mumbai");

                 System.out.println(tm4);
        }
}

// for sorting name in ascending order
class FirstComparator implements Comparator<Employee>
{
        @Override
```

```
        public int compare(Employee e1, Employee e2)
        {
                return e1.name.compareTo(e2.name);
        }
}

// for sorting name in descending order
class SecondComparator implements Comparator<Employee>
{
        @Override
        public int compare(Employee e1, Employee e2)
        {
                return - (e1.name).compareTo(e2.name);
        }
}

// for sorting age in ascending order
class ThirdComparator implements Comparator<Employee>
{
        @Override public int compare(Employee e1, Employee e2)
        {
                return e1.age - e2.age;
        }
}
```

```
// for sorting age in descending order
class FourthComparator implements Comparator<Employee>
{
	@Override public int compare(Employee e1, Employee e2)
	{
		return - (e1.age - e2.age);
	}
}

// Employee class
class Employee
	{
	public int id;
	public String name;
	public Integer age;

	Employee(int id, String name, int age)
	{
		this.id = id;
		this.name = name;
		this.age = age;
	}
```

```
	@Override
	public String toString()
	{
		return " " + this.id + " " + this.name + " "+ this.age;
	}
}
```

IdentityHashMap :

---

public class IdentityHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Clonable, Serializable.

It was introduced from JDK 1.4 onwards.

The IdentityHashMap uses == operator to compare keys and values.

As we know HashMap uses equals() and hashCode() method for comparing objects.

So We should use IdentityHashMap where we need to check the reference instead of logical equality.

HashMap uses hashCode of the "Object key" to find out the bucket loaction, on the other hand IdentityHashMap does not use hashCode() method actually It uses System.identityHashCode(Object o)

```
import java.util.*;

public class IdentityHashMapDemo

{

	public static void main(String[] args)

	{

		HashMap<String,Integer> hm = new HashMap<String,Integer>();


		IdentityHashMap<String,Integer> ihm = new IdentityHashMap<String,Integer>
();


		hm.put("Ravi",23);

		hm.put(new String("Ravi"), 24);


		ihm.put("Ravi",23);

		ihm.put(new String("Ravi"), 24);


		System.out.println("HashMap size :"+hm.size());

		System.out.println("IdentityHashMap size :"+ihm.size());


	}


}
```

WeakHashMap :

---

public WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>

It is a predefined class in java.util package under Map interface.

While working with HashMap, keys of HashMap are of strong reference type. This means the entry of map is not removed by the garbage collector even though the key is set to be null and still it is not eligible for Garbage Collector.

On the other hand while working with WeakHashMap, keys of WeakHashMap are of weak reference type. This means the entry of a map is removed by the garbage collector because it is of weak type.

So, HashMap dominates over Garbage Collector where as Garbage Collector dominates over WeakHashMap.

It contains 4 types of Constructor :

---

1) WeakHashMap wm = new WeakHashMap();

Creates an empty WeakHashMap object with default capacity is 16 and load fator 0.75

2) WeakHashMap wm1 = new WeakHashMap(int initialCapacity);

3) WeakHashMap wm2 = new WeakHashMap(int initialCapacity, float loadFactor);

Eg:- WeakHashMap wm2 = new WeakHashMap(10,0.9);

capacity - The capacity of this map is 10. Meaning, it can store 10 entries.

loadFactor - The load factor of this map is 0.9. This means whenever our hash table is filled by 90%, the entries are moved to a new hash table of double the size of the original hash table.

4) WeakHashMap wm = new WeakHashMap(Map m);

```
import java.util.*;
class WeakHashMapDemo
{
        public static void main(String args[])  throws Exception
        {
                WeakHashMap m = new WeakHashMap();

                Test  t  = new Test();
                m.put(t," Rahul ");  //here we are passing reference 't' as a key

                System.out.println(m);

                t = null;

                System.gc();
```

```
                Thread.sleep(3000);


                System.out.println(m);
        }
}


class Test
{
        public String toString()
        {
                return "demo";
        }


        public void finalize()
        {
                System.out.println("finalize method is called");
        }
}
```

SortedMap :

Elements are sorted based on the key.

Methods of SortedMap :-

---

1) firstKey()

2) lastKey()

3) headMap(Object keyRange)

4) tailMap(Object keyRange)

5) subMap(Object startRange, Object endRange)

```
import java.util.*;
class SortedMapMethodDemo
        {
 public static void main(String args[])
         {
                SortedMap<Integer,String> map=new TreeMap<Integer,String>();
                 map.put(100,"Amit");
                 map.put(101,"Ravi");
                 map.put(102,"Vijay");
                 map.put(103,"Rahul");
                 System.out.println("First Key: "+map.firstKey());
                  System.out.println("Last Key "+map.lastKey());
                  System.out.println("headMap: "+map.headMap(102));
                  System.out.println("tailMap: "+map.tailMap(102));
                  System.out.println("subMap: "+map.subMap(100, 102));
```

```
        }
}
```

Queue interface :-

---

1) It is sub interface of Collection(I)

2) It works in FIFO(First in first out) order.

3) It is an ordered collection.

4) In a queue, insertion is possible from last is called REAR where as deletion is possible from the starting is called FRONT of the queue.

5) From jdk 1.5 onwards LinkedList class implments Queue interface to handle the basic queue operations.

PriorityQueue :-

It is a predefined class in java.util package from Jdk 1.5 onwards.

The LinkedList class has been enhanced to implement Queue interface to handle basic

operation of Queue.

A PriorityQueue is used when we want to store the Objects based on some priority.

The main purpose of PriorityQueue class is to create a "Priority in, Priority out" queue which is opposite to classical FIFO order.

A priority Queue stores the element in a natural sorting order where the elements which are sorted first, will be accessed first.

A priority queue does not permit null elements as well as It uses Comparator interface to sort the elements.

It provides natural sorting order so we can't take non-comparable objects.

The initial capacity of PriorityQueue is 11.

Methods :-

---

offer() :- Used to add an element, It can be also done by add()

poll() :- It is used to fetch the elements from top of the queue, after

fetching it will delete the elements.

peek() :- It is also used to fetch the elements from top of the queue, it will not delete

the elements like poll()

boolean remove(Object element) :- It is used to remove an element. The return type is boolean.

```
import java.util.PriorityQueue;

public class PriorityQueueDemo
{
    public static void main(String[] argv)
    {
        PriorityQueue<String> pq = new PriorityQueue<String>();
        pq.add("2");
        pq.add("4");
            pq.add("6");
        System.out.print(pq.peek() + " "); //2 2 3 4 4
        pq.offer("1");
            pq.offer("9");
        pq.add("3");
        pq.remove("1");
        System.out.print(pq.poll() + " ");
        if (pq.remove("2"))
           System.out.print(pq.poll() + " ");
```

```
        System.out.println(pq.poll() + " " + pq.peek()+"  "+pq.poll());
    }
}
```

```
import java.util.PriorityQueue;


public class PriorityQueueDemo1
{
    public static void main(String[] argv)
    {
        PriorityQueue<String> pq = new PriorityQueue<String>();
        pq.add("9");
        pq.add("8");
                            pq.add("7");
        System.out.print(pq.peek() + " "); //7 3 5 6
        pq.offer("6");
                            pq.offer("5");
        pq.add("3");
        pq.remove("1");    // 5 6 7 8 9
        System.out.print(pq.poll() + " ");
        if (pq.remove("2"))
           System.out.print(pq.poll() + " ");
        System.out.println(pq.poll() + " " + pq.peek());
    }
```

```
}
```

---

classes in java.util package :-

Date :- It is a predefined class available in java.util package to fetch the current system date and time.

```
import java.util.*;

class DateTime

{

	public static void main(String[] args)

	{

		Date d = new Date();

		System.out.println(d);  //d.toString();

	}

}
```

Calendar class :-

It is predefined abstract class in java.util package. It is an abstract class so we can't craete an object but we have a static method inside the Calendar class to get the instance(Object) of Calendar class.

```
Calendar c1 = Calendar.getInstance();
```

It contains get() method where we can pass the final static variables.

Now we have number of predefined final static variables of Calendar class which are as follows :-

1) Calendar.YEAR :- will display the current year

2) Calendar.MONTH :- will display the current month , month starts from

0 (Zero)

3) Calendar.DATE :- will display the current date.

4) Calendar.HOUR :- will display current hour according to the system

5) Calendar.MINUTE :- will display current minute according to the system

---

```
import java.util.*;
public class CalendarTest
{
  public static void main(String[] args)
            {

             Calendar cal = Calendar.getInstance();
             System.out.println("Current Year is :" + cal.get(Calendar.YEAR));
```

```
            System.out.println("Current Month is : " + cal.get(Calendar.MONTH));
            System.out.println("Current Day is : " + cal.get(Calendar.DATE));
            System.out.println("Current Hour is : " + cal.get(Calendar.HOUR));
            System.out.println("Current Minute is : " + cal.get(Calendar.MINUTE));

    }
}
```

```
import java.util.*;
public class CalendarTest1
        {
  public static void main(String[] args)
            {
                    Calendar cal= Calendar.getInstance();
                    System.out.print("The current system date and Time is : " + cal.getTime());
    }
}
```

StringTokenizer :-

It is a predefined class available in java.util package. It is used to break the String into number of pieces , called tokens.

These tokens are stored in StringTokenizer object from where we can fetcg the String using these tokens.

Methods :-

---

1) int countTokens() :-

To count the number of tokens.

2) boolean hasMoreTokens() :-

It will verify the tokens are available or not, if available it will return true if not it will return false.

3) String nextToken() :-

It will fetch the String as a token.

```
import java.util.*;
class STDemo
{
public static void main(String [] args)
        {

                String str ="Hyderabad is a lovely place";
                StringTokenizer st = new StringTokenizer(str,"a");
```

```
                System.out.println("Number of tokens :"+st.countTokens());
                System.out.println("The tokens are :");
                while(st.hasMoreTokens())
                {
                        String token = st.nextToken();
                        System.out.println(token);
                }

        }
}
```

Generic :

Why generic came into picture :

As we know our compiler is known for Strict type checking because java is a strongle typed checked language.

The basic problem with collection is It can hold any kind of Object.

```
ArrayList al = new ArrayList();
al.add("Ravi");
al.add("Aswin");
al.add("Rahul");
```

```
al.add("Raj");

al.add("Samir");


for(int i =0; i<al.size(); i++)

{

  String s = (String) al.get(i);

  System.out.println(s);

}
```

By looking the above code it is clear that Collection stores everything in the form of Object so here even after adding String type only we need type casting as shown below

```
import java.util.*;

class Test1

{

        public static void main(String[] args)

        {

                ArrayList al = new ArrayList();

                al.add("Ravi");

                al.add("Ajay");


                for(int i=0; i<al.size(); i++)

                {

                String name =(String) al.get(i);

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

```
        }
    }
}
```

Even after type casting there is no gurantee that the things that is coming from ArrayList Object is String only because we can add anything in the Collection as a result java.lang.ClassCastException as shown in the program below

```
import java.util.*;
class Test1
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        al.add("Ravi");
        al.add("Ajay");
        al.add(12);

        for(int i=0; i<al.size(); i++)
        {
        String name =(String) al.get(i);
        System.out.println(name);
        }
    }
}
```

To avoid all the above said problem Generics came into picture from JDK 1.5 onwards

-> It deals with type safe Object so there is a gurantee of both the end i.e putting inside and getting out

Advantages :-

---

a) Type safe Object (No compiler warning)

b) Strict compile time checking

c) No need of type casting

```
import java.util.*;
class Test2
{
	public static void main(String[] args)
	{
		ArrayList<String> al = new ArrayList<String>();
		al.add("Ravi");
		al.add("Ajay");
		al.add("Vijay");
```

```
                    //al.add(12);   // It is a compilation error


      for(int i=0; i<al.size(); i++)
                    {
                    String name = al.get(i); //no type casting is required
                    System.out.println(name);
                    }
          }
}
```

```
import java.util.*;
public class Test3
{
public static void main(String[] args)
{
                    ArrayList t = new ArrayList();
                    t.add("alpha");
                    t.add("beta");
                    for (int i = 0; i < t.size(); i++)
                    {
                     String str =(String) t.get(i);
                     System.out.println(str);
                    }
                     t.add(1234);
```

```
                t.add(1256);
                 for (int i = 0; i < t.size(); ++i)
            {
                        Object obj=t.get(i); //we can't perform type casting here
                        System.out.println(obj);
                }
  }
}
```

```
 //Program that describes the return type of any method can be type safe
import java.util.*;
public class Test4
{
        public static void main(String [] args)
        {
                Dog d1 = new Dog();
                Dog d2 = d1.getDogList().get(0);
                System.out.println(d2);
        }
}
class Dog
{
        public List<Dog> getDogList()
```

```
	{
		List<Dog> d = new ArrayList<Dog>();
		d.add(new Dog());
		d.add(new Dog());
		return d;
	}
}
```

Note :- In the above program the compiler will stop us from returning anything which is not compaitable List<Dog> and there is a gurantee that only "type safe list of Dog object" will be returned so we need not to provide type casting as shown below

```
Dog d2 = (Dog) d1.getDogList().get(0);  //before generic.
```

```
import java.util.*;
class Car
{
}
class Test5
{
	public static void main(String [] args)
	{
	ArrayList<Car> a = new ArrayList<Car>();
	a.add(new Car());
```

```
        ArrayList b = a;  //Generic to raw type


        for (Object obj : b)
                System.out.println(obj);
        }
}
```

```
//Mixing generic to non-generic
import java.util.*;
public class Test6
{
        public static void main(String[] args)
        {
                List<Integer> myList = new ArrayList<Integer>();
                myList.add(4);
                myList.add(6);
                myList.add(5);


                UnknownClass u = new UnknownClass();
                int total = u.addValues(myList);
                System.out.println(total);
        }
}
class UnknownClass
```

```
{
        int addValues(List list) //type safe Integer Object  to raw type
        {
        Iterator it = list.iterator();
        int total = 0;
        while (it.hasNext())
        {
        int i = ((Integer)it.next());
        total += i;                     //total = 15
        }
        return total;
        }
}
```

In the above program the compiler will not generate any warning message because even though we are assigning type safe Integer Object to unsafe or raw type List Object but this List Object is not inserting anything in the collection so there is no risk to the caller.

```
//Mixing generic to non-generic
import java.util.*;
public class Test7
{
        public static void main(String[] args)
        {
                List< Integer > myList = new ArrayList< Integer >();
```

```
                myList.add(4);
                myList.add(6);
                UnknownClass u = new UnknownClass();
                int total = u.addValues(myList);
                System.out.println(total);
        }
}
class UnknownClass
{
int addValues(List list)
        {
                list.add(5);        //adding object to raw type
                Iterator it = list.iterator();
                int total = 0;
                while (it.hasNext())
                {
                int i = ((Integer)it.next());
                total += i;
                }
                return total;
        }
}
```

In the above program the compiler will generate warning message because

the unsafe List Object is inserting the Integer object 5 so the type safe Integer objcet is getting

value 5 from unsafe type so there is a problem to the caller method.

By writing ArrayList<Integer> actually JVM does not have any idea that our ArrayList was suppose to hold only Integers.

The all type safe information does not exist at runtime. All our generic code is Strictly for compiler. There is a process done by java compiler called "Type erasure" in which the java compiler converts generic version to non-generic type.

List<Integer> myList = new ArrayList<Integer>();

At the compilation time it is fine but at runtime for JVM the code becomes

List myList = new ArrayList();

Note :- GENERIC IS STRICTLY A COMPILE TIME PROTECTION.

Polymorphism with Array :

```
//Polymorphism with array
import java.util.*;
abstract class Animal
{
        public abstract void checkup();
}
```

```
class Dog extends Animal
{
	public void checkup()
	{
		System.out.println("Dog checkup");
	}
}

class Cat extends Animal
{
	public void checkup()
	{
		System.out.println("Cat checkup");
	}
}

class Bird extends Animal
{
	public void checkup()
	{
		System.out.println("Bird checkup");
	}
}
public class  Test8
```

```
{
	public void checkAnimals(Animal[] animals)
	{
		for(Animal a : animals)
		{
			a.checkup();
		}
	}
	public static void main(String[] args)
	{
		Dog[] dogs={new Dog(), new Dog()};

		Cat[] cats={new Cat(), new Cat(), new Cat()};

		Bird[] birds = {new Bird()};

		Test8 t = new Test8();
		t.checkAnimals(dogs);
		t.checkAnimals(cats);
		t.checkAnimals(birds);
	}
}
```

The polymorphism concept works with array

Polymorphism with generic

```
import java.util.*;
abstract class Animal
{
	public abstract void checkup();
}

class Dog extends Animal
{
	public void checkup()
	{
		System.out.println("Dog checkup");
	}
}

class Cat extends Animal
{
	public void checkup()
	{
		System.out.println("Cat checkup");
	}
```

```
}
class Bird extends Animal
{
	public void checkup()
	{
		System.out.println("Bird checkup");
	}
}
public class Test9
{
	public void checkAnimals(List<Animal> animals)
	{


	}
	public static void main(String[] args)
	{
		List<Dog> dogs = new ArrayList<Dog>();
		dogs.add(new Dog());
		dogs.add(new Dog());

		List<Cat> cats = new ArrayList<Cat>();
		cats.add(new Cat());
		cats.add(new Cat());
```

```
        List<Bird> birds = new ArrayList<Bird>();
        birds.add(new Bird());

        Test9 t = new Test9();
        t.checkAnimals(dogs);
        t.checkAnimals(cats);
        t.checkAnimals(birds);

    }
}
```

So from the above program it is clear that polymorphism does not work in the same way for generics as it does with arrays.

Eg:-

```
Parent [] arr = new Child[5]; //valid
Object [] arr = new String[5]; //valid
```

But in generics the same type is not valid

```
List<Object> list = new ArrayList<Integer>(); //Invalid
List<Number> mylist = new ArrayList<Integer>(); //Invalid
```

```
class Test10
{
public static void main(String [] args)
	{
		Object []obj = new String[3]; //valid
		obj[0] = "Ravi";
		obj[1] = "hyd";
		obj[2] = 12;  //java.lang.ArrayStoreException
	}
}
```

```
import java.util.*;
class Parent
{


}
class Child extends Parent
{
}
class Test11
{
public static void main(String [] args)
	{
		//ArrayList<Parent> lp = new ArrayList<Child>();//error
```

```
                ArrayList<Parent> lp1 = new ArrayList<Parent>();
        }
}
```

Wild card character :

<?> -: Many possibilities

<Dog> -: Only <Dog> can assign

<? super Dog> -: Dog, Animal, Object can assign (Compiler has surity)

<? extends Dog) -: Below of Dog means, sub class of Dog (But the compiler does not have surity because you can have many sub class of Dog in the future, so chances of wrong collections)

//program on wild-card chracter

```
import java.util.*;
class Parent
{

}
class Child extends Parent
{
```

```
}
class Test12
{
public static void main(String [] args)
	{
		List<?> lp = new ArrayList<Child>();
	}
}
```

```
import java.util.*;
class Test13
{
	public static void main(String[] args)
	{

		List<? extends Number> list1 = new ArrayList<Integer>();

		List<? super String> list2 = new ArrayList<Object>();

		List<Integer> list3 = new ArrayList<Integer>();

		List list4 = new ArrayList();

		System.out.println("yes");
```

```
	}
}
```

```
import java.util.*;
class  Test14
{
	public static void main(String[] args)
	{
		try
		{
			List<Object> x = new ArrayList<Object>();
			x.add(10);
			System.out.println(x);
		}
		catch (Exception e)
		{
			System.out.println(e);
		}
	}
}
```

```
class MyClass<T>
{
	T obj;
```

```
	MyClass(T obj)
	{
		this.obj=obj;
	}

	T getobj()
	{
		return obj;
	}
}
class Test15
{
	public static void main(String[] args)
	{
		Integer i=12;
		MyClass<Integer> mi = new MyClass<Integer>(i);
		System.out.println("Integer object stored :"+mi.getobj());

		Float f=12.34f;
		MyClass<Float> mf = new MyClass<Float>(f);
		System.out.println("Float object stored :"+mf.getobj());

		MyClass<String> ms = new MyClass<String>("Rahul");
		System.out.println("String object stored :"+ms.getobj());
```

```
	}
}
--------------------------------------------------------------------------------
class Fruit
{
}
class Apple extends Fruit
{
}
class Basket<E>
{
	private E element;
	public void setElement(E x)
	{
		element = x;
	}
	public E getElement()
	{
		return element;
	}
}

class Test16
{
```

```
	public static void main(String[] args)
	{
		Basket<Fruit> b = new Basket<Fruit>();
		b.setElement(new Apple());

		Apple apple =(Apple) b.getElement();
		System.out.println(apple);

	}
}
```
