ACCESS SPECIFIER

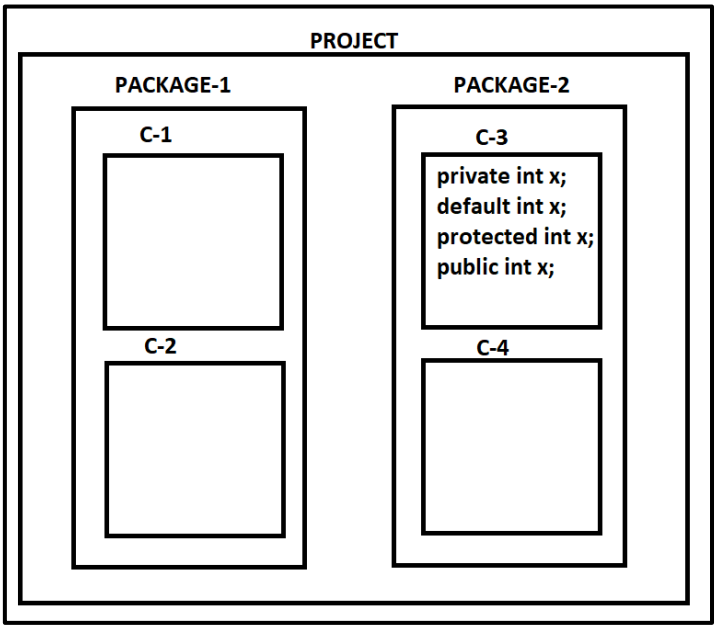
==> In Java we have 4 access specifiers

1.) Private

2.) Default

3.) Protected

4.) Public



Specifier	Package-2		Package-1	
	C3	C4	C1	C2
Private	possible	Not possible	Not possible	Not possible
Default	possible	possible	Not Possible	Not possible
Protected	possible	possible	possible	possible
Public	possible	possible	possible	possible

1.) Private

==> If we declare anything as private, it can be accessed only in the particular area where it is declared.

2.) Default

==> Default can be accessed in the area where it is declared and it can be accessed in other classes of the same package but it cannot be accessed in another packages.

3.) Protected

==> Protected can be accessed all the classes of the same package as well as other packages also.

==> To access in other packages two conditions required

1.) Inheritance 2.) Import

4.) Public

==> Public can be accessed in entire project.

EXAMPLE PROGRAM

```
package package1;

public class Program1

{

    private void m1()

    {

        System.out.println("Private access level is visible");

    }

    void m2()

    {

        System.out.println("Default access level is visible");

    }

    protected void m3()

    {

        System.out.println("Protected access level is visible");

    }

    public void m4()

    {
```

```
        System.out.println("Public access level is visible");
    }
    public static void main(String[] args)
    {
        Program1 p=new Program1();
        p.m1();
        p.m2();
        p.m3();
        p.m4();
    }
}
```

OUTPUT

```
Private access level is visible
Default access level is visible
Protected access level is visible
Public access level is visible
```

EXAMPLE PROGRAM

```
package package2;
public class Program2
{
    public static void main(String[] args)
    {
        Program p=new Program();
        p.m2();
```

```
        p.m3();
        p.m4();
    }
}   // Here m1() is not visible because it is private
```

OUTPUT

Default access level is visible

Protected access level is visible

Public access level is visible

EXAMPLE PROGRAM

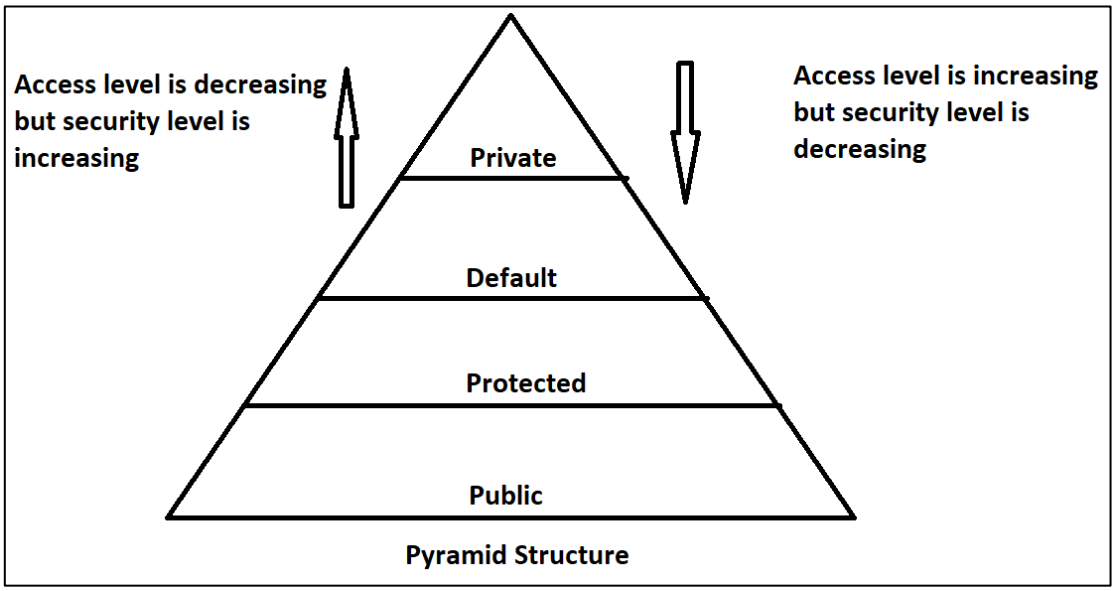
```
package access_specifier_program;
import package1.Program1;
public class Program3 extends Program1
{
    public static void main(String[] args)
    {
        Program1 p=new Program1();
        p.m3();
        p.m4();
    }
}   // Here m1() and m2() is not visible because it is private and default
```

OUTPUT

Protected access level is visible

Public access level is visible

Specifier	Access in Same Class	Access in Another Class of Same Package	Access in Other Class of Other Package
Private	Yes	No	No
Default	Yes	Yes	No
Protected	Yes	Yes	Yes 1.) Inheritance→extends 2.) Import
Public	Yes	Yes	Yes 1.) Import



INBUILT LIBRARIES

==> Inbuilt library is nothing but the collection of inbuilt packages.

==> Inbuilt package is nothing but the collection of inbuilt classes and interfaces.

Package → Class → Methods

Inbuilt Package → Inbuilt Class → Inbuilt Methods

==> We have three very important inbuilt packages present in Java.

1.) java.lang package (*****)

==> There is no explicit import required.

1.) System 2.) Object 3.) String

4.) String Buffer 5.) String Builder 6.) Thread

2.) java.util package

1.) Scanner 2.) Collection

3.) java.io package(File Handling)

1.) File 2.) Buffer Reader 3.) Print Writer

SYSTEM CLASS

==>System is an inbuilt class present in **java.lang** package.

==> Inside System class some methods are present like println(), print(), print(int i), print(byte b), print(short s).

NOTE:

It is not required to import System class explicitly. Because it is present in **java.lang** package.

OBJECT CLASS

==> Object class is the super most class of all the classes present in **java.lang** package.

==> As it is a super class by default all the properties of object class will be inherited implicitly to the sub class.

==> 11 methods are present in object class

1.) toString();**(VVIP)** 2.) equals(Object obj);**(IMP)**

3.) hashCode();**(IMP)** 4.) getClass();

5.) finalize(); 6.) clone();

//Following belongs to ITC (Inter Thread Communication) methods

7.) wait(); 8.) wait(arg);

9.) wait(arg1, arg2); 10.) notify();

11.) notifyAll();

1.) toString()

```
package object_class_programs;

public class ToString_Method {

	public static void main(String[] args) {

		ToString_Method tm1 = new ToString_Method();
		ToString_Method tm2 = new ToString_Method();

		System.out.println(tm1);
		System.out.println(tm1.toString());
		System.out.println(tm2.toString());
	}
}
```

OUTPUT

```
object_class_programs.ToString_Method@15db9742
object_class_programs.ToString_Method@15db9742
object_class_programs.ToString_Method@6d06d69c
```

==> In this example we are using two string method as it is and we are giving complete information of our object as output.

==> toString() is the most important method present in object class.

==> If I print only object reference still I will get complete information of our object because internally toString() get called by object reference.

2.) equals()

```
package object_class_programs;

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

```
		Equals_Method e1 = new Equals_Method();
		Equals_Method e2 = new Equals_Method();

		System.out.println(e1);
		System.out.println(e2);

		boolean b = e1.equals(e2);

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

OUTPUT

```
object_class_programs.Equals_Method@15db9742
object_class_programs.Equals_Method@6d06d69c
false
```

==> In our example we are using equals() to perform comparison in between two objects.

==>Here I am getting output as false because given object and current object is performing address level comparison.

3.) hashCode()

```
package object_class_programs;

public class HashCode_Method {

	public static void main(String[] args) {

		HashCode_Method h1= new HashCode_Method();
		System.out.println(h1);

		int i = h1.hashCode();
		System.out.println(i);
	}
}
```

OUTPUT

```
object_class_programs.HashCode_Method@15db9742
366712642
```

==> Hash code is nothing but unique identity of an object.

==> Hash code should be universal and unique.

==> In the example we are getting unique identity of our object.

//Example

```
package object_class_programs;

public class HashCode_Method1 {
public static void main(String[] args) {

		HashCode_Method1 h1 = new HashCode_Method1();
		HashCode_Method1 h2 = new HashCode_Method1();
		System.out.println(h1);
		System.out.println(h2);

		int i = h1.hashCode();
		int j = h2.hashCode();
		System.out.println(i);
		System.out.println(j);
		System.out.println(i==j);
	}
}
```

OUTPUT

```
object_class_programs.HashCode_Method1@15db9742
object_class_programs.HashCode_Method1@6d06d69c
366712642
1829164700
false
```

//EXAMPLE

```
package object_class_programs;

public class Car {

	private String bname;
	private int speed;
	private String color;
	private int mileage;
```

```
	public Car(String bname, int speed, String color, int mileage)
	{
		this.bname=bname;
		this.speed=speed;
		this.color=color;
		this.mileage=mileage;
	}

	public String toString()
	{
		return "brand name is : "+ bname + " [ color is : ]"+color;
	}

}
package object_class_programs;

public class CarDetails {

	public static void main(String[] args) {

		Car c = new Car("TATA",60,"BLACK",12);
		Car c1 = new Car("AUDI",120,"RED",8);
		System.out.println(c);
		System.out.println(c1);
	}
}
```

OUTPUT

```
brand name is : TATA [ color is : ]BLACK
brand name is : AUDI [ color is : ]RED
```

==> All the objects must have useful information.

==> But it is not mandatory that only one useful information will be present.

==> Multiple useful information can be present.

//EXAMPLE //Overriding of hashCode in Car

```
package object_class_programs;

public class Car {
```

```
	private String bname;
	private int speed;
	private String color;
	private int mileage;

	public Car(String bname, int speed, String color, int mileage)
	{
		this.bname=bname;
		this.speed=speed;
		this.color=color;
		this.mileage=mileage;
	}

	public String toString()
	{
		return "brand name is : "+ bname + "[color is :]"+color;
	}

	public int hashCode()
	{
		return speed;
	}
}

package object_class_programs;

public class CarDetails {

	public static void main(String[] args) {

		Car c = new Car("TATA",60,"BLACK",12);
		Car c1 = new Car("AUDI",120,"RED",8);
		System.out.println(c);
		System.out.println(c1);

		int i = c.hashCode();
		int j = c1.hashCode();

		System.out.println(i);
		System.out.println(j);
	}
}
```

OUTPUT

```
brand name is : TATA[color is :]BLACK
brand name is : AUDI[color is :]RED
60
120
```

//EXAMPLE

//Overriding of hashCode in Employee

```
package object_class_programs;

public class Employee {

	private String ename;
	private int empId;
	private double salary;

	public Employee(String ename, int empId, double salary)
	{
		this.ename = ename;
		this.empId = empId;
		this.salary = salary;
	}
	public String toString()
	{
		return "emp name : "+ename;
	}
	public int hashCode()
	{
		return empId;
	}

}
package object_class_programs;

public class EmployeeDetails {

	public static void main(String[] args) {

		Employee e = new Employee("RAJ", 12, 25000.00);
		Employee e1 = new Employee("RAM", 15, 30000.00);
```

```
            System.out.println(e);
            System.out.println(e1);

            int i = e.hashCode();
            int j = e1.hashCode();
            System.out.println(i);
            System.out.println(j);
        }
}
```

OUTPUT

```
emp name : RAJ
emp name : RAM
12
15
```

//EXAMPLE //Overriding of hashCode in Animal

```
package object_class_programs;

public class Animal {

        private String aName;
        private int aAge;
        private String color;

        public Animal(String aName, int aAge, String color) {

                this.aName = aName;
                this.aAge = aAge;
                this.color = color;
        }

        @Override
        public String toString() {
                return "Animal [aName=" + aName + ", aAge=" + aAge + ",
color=" + color + "]";
        }

        @Override
        public int hashCode() {
                return aAge;
        }
```

```
}
```

```
package object_class_programs;

public class AnimalDetails {

	public static void main(String[] args) {

		Animal a = new Animal("TIGER", 12, "YELLOW");
		Animal a1 = new Animal("COW", 8, "WHITE");
		System.out.println(a);
		System.out.println(a1);

		int i = a.hashCode();
		int j = a1.hashCode();
		System.out.println("Age is "+i);
		System.out.println("Age is "+j);
	}
}
```

OUTPUT

```
Animal [aName=TIGER, aAge=12, color=YELLOW]
Animal [aName=COW, aAge=8, color=WHITE]
Age is 12
Age is 8
```

Q. We have '==' operator but why do we need equals method?

A: With the help of '==' operator we can perform comparison in between two objects. But the comparison will be address level comparison, to get conten level comparison we need to override.

==> But java does not support operator overriding.

That's why we need equals() method. Bexcause method overriding is possible in Java.

//EXAMPLE

```
package object_class_programs;

public class Comparison {
```

```
	private double salary;

	public Comparison(double salary)
	{
		this.salary=salary;
	}
	public static void main(String[] args) {

		Comparison c = new Comparison(25000);
		Comparison c1 = new Comparison(25000);
		System.out.println(c);
		System.out.println(c1);
		System.out.println(c==c1);
	}

}
```

OUTPUT

```
object_class_programs.Comparison@15db9742
object_class_programs.Comparison@6d06d69c
false
```

==> In this example we are getting output false because address level comparison is done by '==' operator

//EXAMPLE

```
package object_class_programs;

public class ComparisonNew {

	private int age;

	public ComparisonNew(int age){

		this.age=age;
	}
	public static void main(String[] args) {

		ComparisonNew cn = new ComparisonNew(20);
		ComparisonNew cn1 = new ComparisonNew(30);
```

```
		System.out.println(cn.equals(cn1));
	}

	public boolean equals(Object obj){

		ComparisonNew cn2 = (ComparisonNew) obj;
		return this.age == cn2.age;
	}

}
```

OUTPUT

```
false
```

//EXAMPLE

```
package object_class_programs;

public class EmployeeNew {

	private String eName;
	private int eId;
	private double salary;

	public EmployeeNew(String eName, int eId, double salary) {

		this.eName = eName;
		this.eId = eId;
		this.salary = salary;
	}

	@Override
	public String toString() {
		return "EmployeeDetails [eName=" + eName + "]";
	}

	@Override
	public int hashCode() {

		return eId;
	}

	public boolean equals(Object obj){
```

```
            EmployeeNew en2 = (EmployeeNew) obj;
            /*return this.salary == en2.salary;*/
            double i = this.salary;
            double j = en2.salary;
            return i==j;
        }
}
package object_class_programs;

public class EmployeeNewDetails {

        public static void main(String[] args) {

            EmployeeNew en = new EmployeeNew("RAJA", 7, 25000);
            EmployeeNew en1 = new EmployeeNew("RAMA", 12, 45000);
            System.out.println(en);
            System.out.println(en1);

            int i = en.hashCode();
            int j = en1.hashCode();
            System.out.println("Emp Id is "+i);
            System.out.println("Emp Id is "+j);

            boolean b = en.equals(en1);
            System.out.println(b);
        }
}
```

OUTPUT

```
EmployeeDetails [eName=RAJA]
EmployeeDetails [eName=RAMA]
Emp Id is 7
Emp Id is 12
false
```

//EXAMPLE ASSIGNMENT

```
package object_class_programs;

public class Player {
```

```
	private String pName;
	private int pAge;
	private int pRuns;

	public Player(String pName, int pAge, int pRuns) {
		this.pName = pName;
		this.pAge = pAge;
		this.pRuns = pRuns;
	}

	@Override
	public String toString() {
		return "Player Name [pName=" + pName + "]";
	}

	@Override
	public int hashCode() {

		return pAge;
	}

	public boolean equals(Object obj)
	{
		Player p2 = (Player) obj;
		int i= this.pRuns;
		int j = p2.pRuns;
		return i==j;
	}
}

package object_class_programs;

public class PlayerDetails {

	public static void main(String[] args) {

		Player p = new Player("DHONI", 45, 100);
		Player p1 = new Player("VIRAT", 40, 80);
		System.out.println(p);
		System.out.println(p1);

		int i = p.hashCode();
		int j = p1.hashCode();
```

```
            System.out.println("Player Age : "+i);
            System.out.println("Player Age : "+j);

            boolean b = p.equals(p1);
            System.out.println(b);
        }

}
```

OUTPUT

```
Player Name [pName=DHONI]
Player Name [pName=VIRAT]
Player Age: 45
Player Age: 40
false
```

STRING CLASS

==> String is an inbuilt class.

==> String class is present in java.lang package.

==> String class is a final class.

==> Inheritance is not possible.

==> String class don't have any sub class.

==>We can create object of String class in two ways

1.) With new keyword

2.) Without new keyword (or) With Literals

Object Creation with new keyword

==> Whenever we are creating String object with new keyword object will be created in String Non Constant Pool Area.

Syntax: String s = new String("Hello");

String s1 = new String("Hello");

==> Inside String Non Constant Pool Area two objects will be created because String Non Constant Pool Area allows duplicate objects.

Object Creation without new keyword (or) With Literals

==> Whenever we are creating String object without new keyword object will be created in String Constant Pool Area.

==> If we create two objects with same content both the object references will go to the same object because String Constant Pool Area doesn't allow duplicate objects.

```
package string_programs;

public class String_Const {

	public static void main(String[] args) {

		String s1 = new String("Hello");
		String s2 = new String("Hello");
		String s3 = "Hii";
		String s4 = "Hii";

		System.out.println(s1==s2);
		System.out.println(s3==s4);
	}
}
```

OUTPUT

```
false
true
```

String Immutable Property

==> String objects are immutable in nature.

==> Immutable means which cannot be changed.

==> If we try to change we will get another object with the old reference.

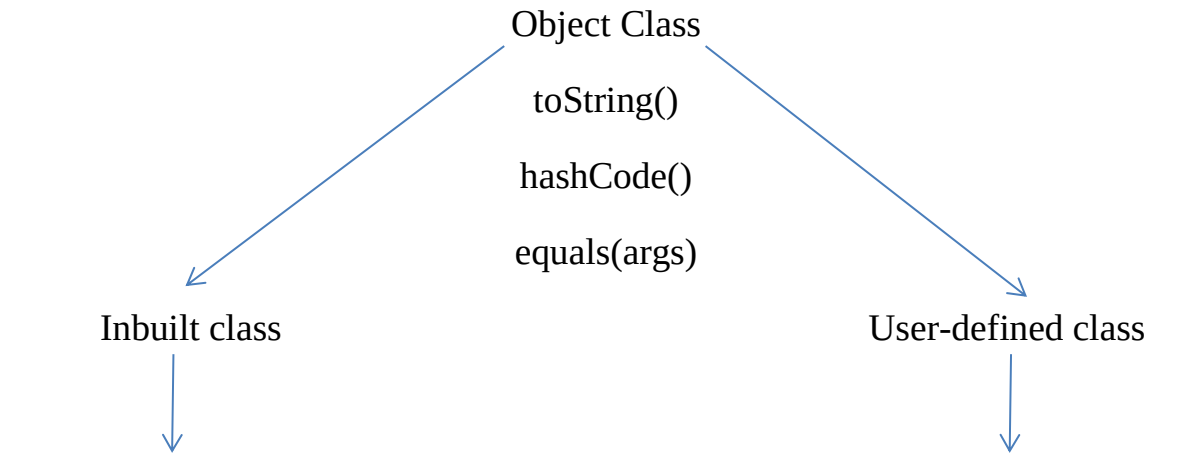
==> The old reference will be removed from the old object.

```
package string_programs;

public class String_Immutable {
```

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

                String s1 = new String("Hello");
                String s2 = new String("Hello");

                String s3 = "Hii";
                String s4 = "Hii";

                System.out.println(s1==s2);
                System.out.println(s3==s4);

                s4 = "Bye";
                System.out.println(s3==s4);
        }

}
```

OUTPUT

```
false
true
false
```



It will automatically override internally and give content level output

It will give address level output to get content level output we have to override.

ARRAYS

==> Array is an object.

==> Array is the collection of similar datatype.

==> Array is homogenous in nature.

==> It is fixed in size.

==> Based on Dimension, we have 2 types.

1.) Single Dimension

2.) Two Dimension (2D)

==> Based on Datatype, we have 2 types.

1.) Primitive Datatype

2.) Non Primitive Datatype

==> Based on Object Creation, we have 2 types.

1.) With new

2.) Without new

Single Dimensional Primitive Array With Using NEW Keyword

Syntax: datatype [] reference_var = new datatype[size];

Eg: int []x = new int[4];

Accessing Array Primitive

```
package array_programs;

public class Accessing {

	public static void main(String[] args) {
		// Creation of array
		int []x = new int[4];

		// Adding array elements
		x[0] = 10;
		x[1] = 20;
```

```
                x[2] = 30;
                x[3] = 40;

                // Fetching the elements
                System.out.println("Array Elements are ");
                for(int i=0; i<=x.length-1; i++)
                {
                        System.out.println(x[i]);
                }
        }
}
```

OUTPUT

```
Array Elements are
10
20
30
40
```

Accessing Array Non-Primitive

```
package array_programs;

public class AccessingNPDT {

        public static void main(String[] args) {

                String []s = new String[4];

                s[0] = "Raj";
                s[1] = "Maharaj";
                s[2] = "Yamraj";
                s[3] = "Nagraj";

                System.out.println("Array Elements are ");
                for(int i=0; i<=s.length-1; i++)
                {
                        System.out.println(s[i]);
                }
        }
}
```

OUTPUT

Array Elements are
Raj
Maharaj
Yamraj
Nagraj

Q. WAP FOR ELEMENT PRESENT AT 'I'TH INDEX

```
package array_programs;

import java.util.Scanner;

public class ArrayIndex {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Array Size : ");
		int size = sc.nextInt();
		int []x = new int[size];
		for(int i=0; i<=x.length-1; i++)
		{
			System.out.println("Enter the Element @ index : "+i);
			x[i] = sc.nextInt();
		}
		for(int i=0; i<=x.length-1; i++)
		{
			System.out.println("Element present in "+i+" index is
"+x[i]);
		}
	}
}
```

OUTPUT

Enter Array Size : 5
Enter the Element @ index : 0
2
Enter the Element @ index : 1
3
Enter the Element @ index : 2
4

```
Enter the Element @ index : 3
5
Enter the Element @ index : 4
6
Element present in 0 index is 2
Element present in 1 index is 3
Element present in 2 index is 4
Element present in 3 index is 5
Element present in 4 index is 6
```

Q. WAP TO PRINT ARRAY ELEMENTS FROM USER. WE HAVE TO TAKE 6 DOUBLE ARRAY ELEMENTS.

```
package array_programs;

import java.util.Scanner;

public class DoubleArray {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Array Size :");
		int size = sc.nextInt();
		double []x = new double[size];
		for(int i=0; i<=x.length-1; i++)
		{
			System.out.println("Enter the Element @ index : "+i);
			x[i] = sc.nextDouble();
		}
		for(int i=0; i<=x.length-1; i++)
		{
			System.out.println("Element present in "+i+" index is
"+x[i]);
		}
	}
}
```

OUTPUT

```
Enter Array Size :
3
```

```
Enter the Element @ index : 0
12.25
Enter the Element @ index : 1
22.50
Enter the Element @ index : 2
33.75
Element present in 0 index is 12.25
Element present in 1 index is 22.5
Element present in 2 index is 33.75
```

Q. WAP AND TAKE 5 STRINGS AS YOUR FAMILY MEMBERS.TAKE INPUT FROM USER AND PRINT ALL THE FAMILY MEMBERS NAME.

```
package array_programs;

import java.util.Scanner;

public class StringFamily {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Array Size :");
		int size = sc.nextInt();
		String []s = new String[size];
		for(int i=0; i<=s.length-1; i++)
		{
			System.out.println("Enter Family Members Name @ index :
"+i);
			s[i] = sc.next();
		}
		for(int i=0; i<=s.length-1; i++)
		{
			System.out.println("Enter Family Members are: ");
		}
		for(int i=0; i<=s.length-1; i++)
		{
			System.out.println("Element present in "+i+" index is
"+s[i]);
		}
	}
```

}

OUTPUT

```
Enter Array Size :
2
Enter Family Members Name @ index : 0
Aswin
Enter Family Members Name @ index : 1
Prakash
Enter Family Members are:
Enter Family Members are:
Element present in 0 index is Aswin
Element present in 1 index is Prakash
```

Q. WAP TO CREATE AN ARRAY WITH 5 INTEGERS ELEMENTS AND PRINT IN REVERSE ORDER

```
package array_programs;

import java.util.Scanner;

public class ReverseArray {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Array Size : ");
		int size = sc.nextInt();
		int []x = new int[size];
		for(int i=0; i<=x.length-1; i++)
		{
			System.out.println("Enter the Element @ index : "+i);
			x[i] = sc.nextInt();
		}
		for(int i=x.length-1; i>=0; i--)
		{
			System.out.println("Element present in "+i+" index is
"+x[i]);
		}
	}
}
```

OUTPUT

```
Enter Array Size :
5
Enter the Element @ index : 0
2
Enter the Element @ index : 1
3
Enter the Element @ index : 2
4
Enter the Element @ index : 3
5
Enter the Element @ index : 4
6
Element present in 4 index is 6
Element present in 3 index is 5
Element present in 2 index is 4
Element present in 1 index is 3
Element present in 0 index is 2
```

Single Dimensional Primitive Array Without Using NEW Keyword

```
package array_programs;

public class SPAWithoutNew {

	public static void main(String[] args) {

		int []x = {30,40,50,60,70};
		for(int i=x.length-1 ; i>=0 ; i--)
		{
			System.out.println(x[i]);
		}
	}
}
```

OUTPUT

```
70
60
50
40
30
```

Q. WAP FOR ARRAY AND FETCH THE ELEMENTS WITH THE HELP OF FOR EACH LOOP

```
package array_programs;

public class ForEach {

	public static void main(String[] args) {

		int []x = {40,50,60,70};
		for(int i:x)
		{
			System.out.println(i);
		}
	}
}
```

OUTPUT

```
40
50
60
70
```

Q. WAP TO PRINT THE HIGHEST ELEMENT PRESENT IN ARRAY

```
package array_programs;

public class HighestElement {

	public static void main(String[] args) {

		int []x = {60,40,50,70,80,55};
		int highest = x[0];
		for(int i:x)
		{
			if(i>highest)
			{
				highest = i;
			}
		}
		System.out.println("Highest Element is "+highest);
	}
}
```

OUTPUT

Highest Element is 80

Q. WAP TO PRINT THE LOWEST ELEMENT PRESENT IN ARRAY

```
package array_programs;

public class LowestElement {

	public static void main(String[] args) {

		int []x = {60,40,50,70,80,55};
		int lowest = x[0];
		for(int i:x)
		{
			if(i<lowest)
			{
				lowest = i;
			}
		}
		System.out.println("Lowest Element is "+lowest);
	}
}
```

OUTPUT

Lowest Element is 40

Q. WAP TO PRINT THE HIGHEST AND LOWEST ELEMENT PRESENT IN ARRAY

```
package array_programs;

public class HighLowElement {

	public static void main(String[] args) {

		int []x = {60,40,50,70,80,55};
		int highest = x[0];
		int lowest = x[0];
		for(int i:x)
		{
```

```
                if(i>highest)
                {
                        highest = i;
                }
                if(i<lowest)
                {
                        lowest = i;
                }
        }
        System.out.println("Highest Element is "+highest);
        System.out.println("Lowest Element is "+lowest);
    }
}
```

OUTPUT

```
Highest Element is 80
Lowest Element is 40
```

EXCEPTION HANDALING:

==> In java there are two statements
    1. Normal statement 2. Dangerous statement
==> If Dangerous statement behaves normally, there is no problem.
==> If Dangerous statement behaves abnormally, exception will occur.

Exception

==> Exception is nothing but run time failure of our java program during execution if JVM finds dangerous statement behaves abnormally immediately it will point out that exception and terminates the code at that moment only so that further part of that code will not get executed

Example :

```
    int i = 10;
    int j = 0;      //Arithmetic Exception
    int k = i/j;
    System.out.println(k);
```

Program

```
package exception_handling_programs;

public class Exception_Main {
```

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

		System.out.println("Main Starts");
		int i = 10;
		int j = 0;
		int k = i/j;
		System.out.println(k);
		System.out.println("Main Ends");
	}
}
```

OUTPUT

```
Main Starts
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at
exception_handling_programs.Exception_Main.main(Exception_Main.java:10)
```

Q. How to handle Exception?

A: With the help of try and catch block we can handle Exception.

==> Inside try block we will keep dangerous statement and inside catch block we will handle Exception.

```
package exception_handling_programs;

public class TryCatch_Block {

	public static void main(String[] args) {

		System.out.println("Main Starts");
		try
		{
			int i = 10;
			int j = 0;
			int k = i/j;
			System.out.println(k);
		}
		catch(ArithmeticException e)
		{
			System.out.println("Exception Handled");
		}
```

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

OUTPUT

```
Main Starts
Exception Handled
Main Ends
```

Case-1: One Dangerous statement can have Multiple Exception

```
package exception_handling_programs;

public class ArrayIndexException {

        public static void main(String[] args) {

                try
                {
                        int []x = new int[3];
                        x[0] = 10;
                        x[1] = 0;
                        x[2] = 40;
                        x[3] = 50;
                        int z = x[0]/x[1];
                        System.out.println(x);
                }
                catch (ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("Index Exception Handled");
                }
                catch (ArithmeticException a)
                {
                        System.out.println("Arithmetic Exception Handled");
                }
        }
}
```

OUTPUT

```
Index Exception Handled
```

Ways of Try-Catch Block

1. Possible Way

   try

   { }

   catch(Exception_type)

   { }

2. Possible Way

   try

   { }

   catch(Exception_type)

   { }

   catch(Exception_type)

   { }

3. Not Possible Way

   try

   { }

   try

   { }

   catch(Exception_type)

   { }

Finally

==> Finally is the special type of block which will execute all the time.

==> If exception generates also finally block will be executed.

==> That’s why we will keep all our costly resources inside finally block.

Eg: Database

```
package exception_handling_programs;

public class FinallyBlock {

	public static void main(String[] args) {

		System.out.println("Main Starts");
		try
		{
			int i = 10;
			int j = 0;
			int k = i/j;
			System.out.println(k);
		}
		finally
		{
			System.out.println("My Databse is Safe");
		}
		System.out.println("Main Ends");
	}
}
```

OUTPUT

```
Main Starts
My Databse is Safe
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exception_handling_programs.FinallyBlock.main(FinallyBlock.java:13)
```

Ways of Try-Catch Block

1. Possible Way

   try

   { }

   finally

   { }

2. Possible Way

   try

   { }

   catch(Exception_type)

   { }

   finally

   { }

3. Not Possible Way

   try

   { }

   finally

   { }

   catch(Exception_type)

   { }

COMPILE TIME EXCEPTION

==> The Exception which generates during compilation is called Compile Time Exception.

==> These Exceptions are checked by compiler, that's why it is also called as Check Exception.

==> We can handle Compile Time Exception in 2 ways.

1. Surrounded by try-catch
2. With the help of throws keyword

```
package exception_handling_programs;

public class CTime_Exception {

    public static void main(String[] args) throws InterruptedException {
```

```
                System.out.println("Main Starts");
                Thread.sleep(3000);
                System.out.println("Main Ends");
                Thread.sleep(2000);
                m1();
        }
        public static void m1() throws InterruptedException
        {
                System.out.println("M1 Starts");
                Thread.sleep(3000);
                System.out.println("M1 Ends");
        }
}
```

OUTPUT

```
Main Starts
Main Ends
M1 Starts
M1 Ends
```

```
package exception_handling_programs;

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

                System.out.println("Main Starts");
                try
                {
                        Thread.sleep(5000);
                }
                catch (InterruptedException e)
                {
                }
                System.out.println("Main Ends");
        }
}
```

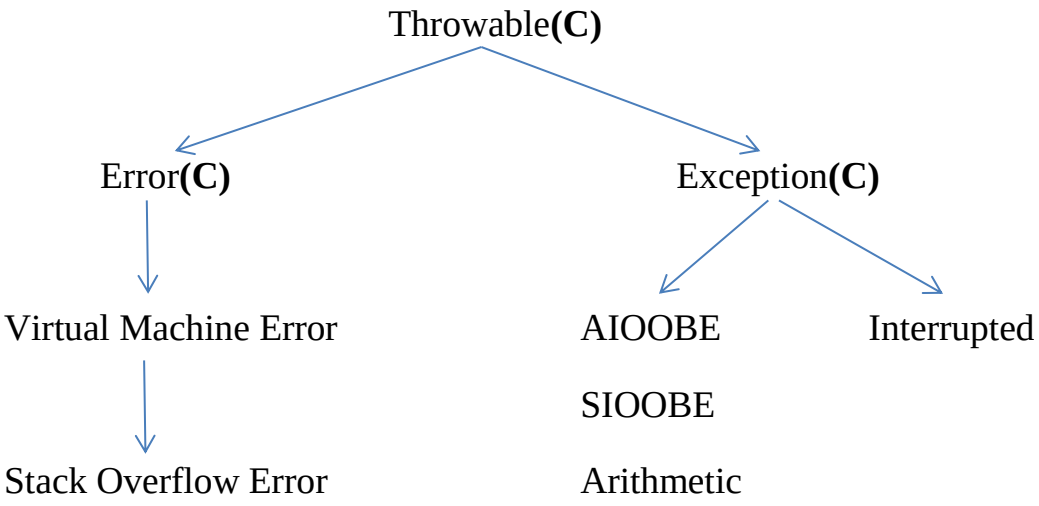
OUTPUT

```
Main Starts
Main Ends
```

DIFFERENCE BETWEEN COMPILE TIME AND RUN TIME EXCEPTION

COMPILE TIME EXCEPTION	RUN TIME EXCEPTION
The Exceptions which generate during compilation is called Compile Time Exception	The Exceptions which generate during execution is called Run Time Exception
Compile Time Exceptions are checked by compiler, that's why it is called Checked Exception.	Run Time Exceptions are pointed by JVM, that's why it is called Unchecked Exception.
Two ways are there to handle Compile Time Exception 1. Surrounded by try-catch 2. With the help of throws keyword	Only one way to handle Run Time Exception 1. try-catch

GENERIC EXCEPTION HANDLING



AIOOBE - **Array Index Out Of Boundary Exception**

==> Throwable is the super most class of all the error and exception class.

==> With the help of Throwable class we can perform Generic Exception Handling.

```
package exception_handling_programs;

public class ThrowableException {

    public static void main(String[] args) throws InterruptedException {
```

```
		System.out.println("Main Starts");
		Thread.sleep(3000);
		System.out.println("Main Ends");
		Thread.sleep(3000);
		test();
	}
	public static void test() throws InterruptedException
	{
		System.out.println("Test fails");
		Thread.sleep(3000);
		System.out.println("Test Pass");
	}
}
```

OUTPUT

```
Main Starts
Main Ends
Test fails
Test Pass
```

Q.) Is it mandatory to handle Exception all the time?

==> No, It is not mandatory to handle exception all the time.

==> It depends on application business purpose.

CUSTOMIZED EXCEPTION

==> With the help of throw keyword we can perform (or) create our own exception. This is nothing but called as Customized Exception.

```
package exception_handling_programs;

public class CustomizedException {

	public void legalAge(int age) throws Exception
	{
		if(age>=18)
		{
			System.out.println("Age is Legal!!!");
		}
		else
```

```
            {
                throw new Exception("Age is not Legal!!!");
            }
        }
        public static void main(String[] args) throws Exception {

            CustomizedException ce = new CustomizedException();
            ce.legalAge(17);
        }
}
```

OUTPUT

```
Exception in thread "main" java.lang.Exception: Age is not Legal!!!
        at exception_handling_programs.CustomizedException.legalAge(CustomizedException.java:13)
        at exception_handling_programs.CustomizedException.main(CustomizedException.java:19)
```

DIFFERENCE BETWEEN THROW,THROWS AND THROWABLE

THROW	THROWS	THROWABLE
throw is a keyword	throws is a keyword	Throwable is the super most class of exception and error.
With the help of throw keyword we can create our own Exception	With the help of throws keyword we can handle Compile Time Exception.	With the help of Throwable class we can perform Generic Exception Handling

FINAL	FINALLY	FINALIZE
final is a keyword	finally is a block.	Finalize is a method
==> If we declare final with variable it cannot be reinitialized. ==> If we declare final with method it cannot be overrided. ==> If we declare final with class it cannot be perform inheritance.	finally block will be executed all the time	With the help of finalize method we can remove the external dependency from a dereffered object.

WRAPPER CLASS

Q. Is Java 100% Object Oriented Programming Language?

A: Java is not 100% Object Oriented Programming Language. To make Java 100% Object Oriented Programming Language wrapper class concept introduced in Java in the version "**JDK1.1"**

Q. Why Java was not 100% Object Oriented Programming Language?

A: Because of Primitive Datatype.

Primitive Datatype	Wrapper Class Object
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
Double	Double
Char	Character
Boolean	Boolean

BOXING

==> Converting Primitive Datatype to Wrapper Class Object is called Boxing.

AUTO-BOXING

==> Auto-Boxing is an implicit process by which we can convert primitive datatype to wrapper class object.

Example:

```
package wrapper_class_programs;

public class BoxingProcess {

	public static void main(String[] args) {

		int i = 10;
		//Boxing
		Integer a = new Integer(i);
		/*Auto-Boxing
		Integer a = i;*/
```

```
            System.out.println(i);
            System.out.println(a);
        }
}
```

OUTPUT

```
10
10
```

AUTO-UNBOXING

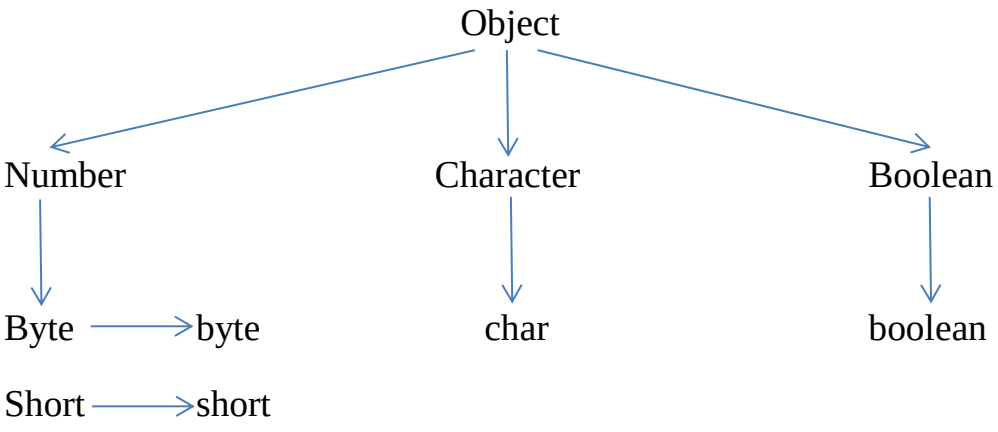
==> Converting Wrapper class object to Primitive Datatype is called Auto UnBoxing.

Example:

```
package wrapper_class_programs;

public class UnBoxingProcess {

        public static void main(String[] args) {

            int i = 10;

            Integer a = i;
            int z = a;
            System.out.println(z);
        }
}
```

OUTPUT

```
10
```



Integer ⟶ int

Long ⟶ long

Float ⟶ float

Double ⟶ double

Example:

1.) int i = 10; int ⟶ 10

2.) Integer a = 10;

int ⟶ Integer (**Boxing**)

3.) Number n = 10;

int ⟶ Integer (**Boxing**) ⟶ Number(**Upcasting**)

4.) Object obj;

int ⟶ Integer (**Boxing**) ⟶ Number(**Upcasting**)

↓

Object(**Upcasting**)

5.) int i = 'a';

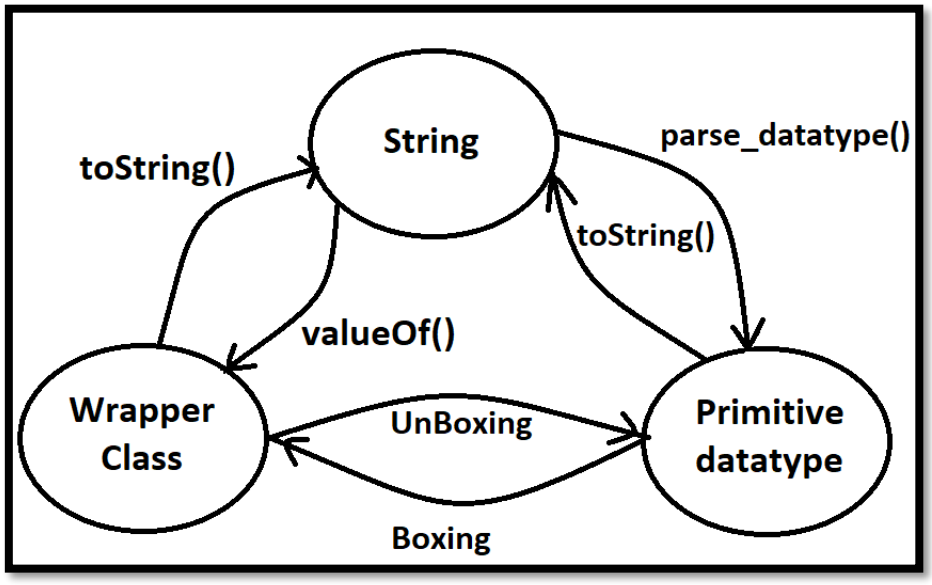
int ⟶ Integer (**Boxing**) ⟶ Number(**Upcasting**)

↓

char(**UnBoxing**) ⟵ Character(**Downcasting**) ⟵ Object(**Upcasting**)

In the above 5th example Object ⟶ char is not possible.

IMPORTANT CONVERSION



Example:

```
package wrapper_class_programs;

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

			/*Auto-Boxing*/
		int x = 10;
		Integer y = x;
		System.out.println("Boxing : "+y);
			/*Auto-UnBoxing*/
		int z = y;
		System.out.println("Un-Boxing : "+z);
			/*Wrapper Class to String*/
		Integer a = 20;
		String s = a.toString();
		System.out.println("Wrapper Class to String : "+(s+20));

			/*Primitive Datatype to String*/
		int k = 50;
		String s1 = Integer.toString(x);
		System.out.println("Primitive Datatype to String : "+(s1+40));

			/*String to WrapperClass*/
		String s2 = "60";
		Integer l = Integer.valueOf(s2);
		System.out.println("String to Wrapper Class : "+(l+50));

			/*String to Primitive Datatype*/
		String s3 = "100";
		int g = Integer.parseInt(s3);
		System.out.println("String to Primitive Datatype : "+(g+100));
	}
}
```

OUTPUT

```
Boxing : 10
Un-Boxing : 10
Wrapper Class to String : 2020
Primitive Datatype to String : 1040
String to Wrapper Class : 110
String to Primitive Datatype : 200
```

COLLECTION

==> Collection is an Interface present in **java.util** package.

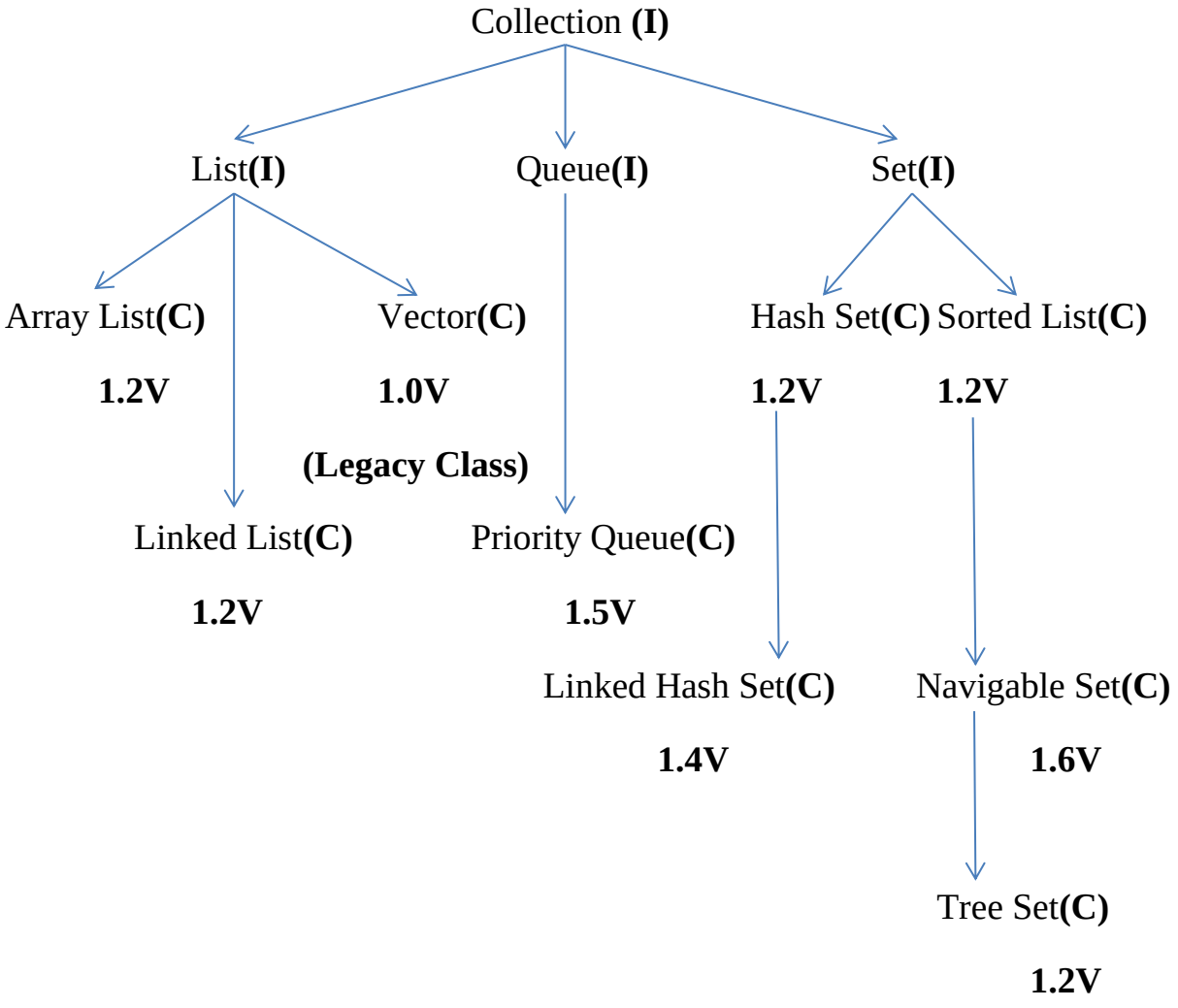
==> To remove the drawbacks of array, Collection concept introduced in Java.

==> Collection is Homogeneous as well as Heterogeneous in nature.

==> It is not fixed in size, it is autogrowable in nature.

==> In Collection inbuilt method library is present.

ARRAY	COLLECTION
Array is an object.	Collection is an interface.
Array is Homogeneous in nature.	Collection is Homogeneous as well as Heterogeneous in nature.
Array is fixed in size.	Collection is autogrowable in nature.
In Array no inbuilt method library is present.	In Collection inbuilt method library is present.



VECTOR

==> Vector is introduced in the version **JDK1.0**. That's why it is called as Legacy class.

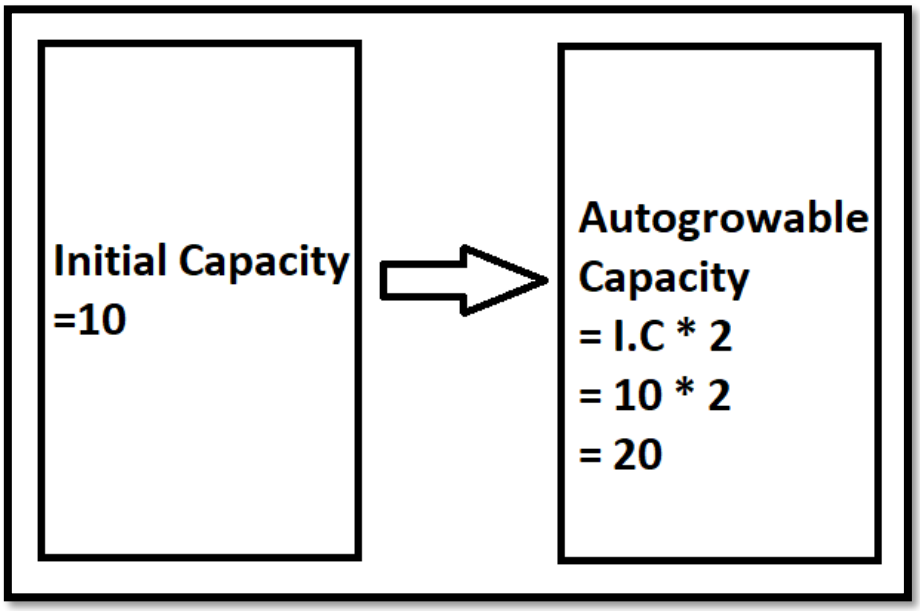
==> Vector is Homogeneous as well as Heterogeneous in Nature.

==> Insertion order is preserved.

==> Vector allows duplicate values.

==> Vector allows NULL values.

==> Initial capacity is 10, Autogrowable capacity is double of its initial capacity.




```
package collection_programs;
/*Step-2 : Import from java.util Package*/
import java.util.Vector;

public class CollVector {

	public static void main(String[] args) {

		/*Step-1 : Creation of Implementation class Object*/
		Vector v = new Vector();

		/*Adding Elements*/
		v.add(50);
```

```
                v.add(97);
                v.add("pushpa");
                v.add(24.56);
                v.add(50);
                v.add(null);

                /*Fetch all the Elements at a time*/
                System.out.println(v);

                /*Searching Elements*/
                System.out.println(v.contains("pushpa"));

                /*Fetching a Particular Element*/
                System.out.println(v.get(2));

                /*Deletion Operation*/
                System.out.println(v.remove(2));
                System.out.println(v);

                /*Modification of Particular Cell*/
                v.set(4,"Kattappa");
                System.out.println(v);

                /*Modification of Entire Vector*/
                v.add(1,"Bahubali");
                System.out.println(v);
        }
}
```

OUTPUT

```
[50, 97, pushpa, 24.56, 50, null]
true
pushpa
pushpa
[50, 97, 24.56, 50, null]
[50, 97, 24.56, 50, Kattappa]
[50, Bahubali, 97, 24.56, 50, Kattappa]
```

ARRAYLIST

==> It is introduced in the version **JDK1.2**.

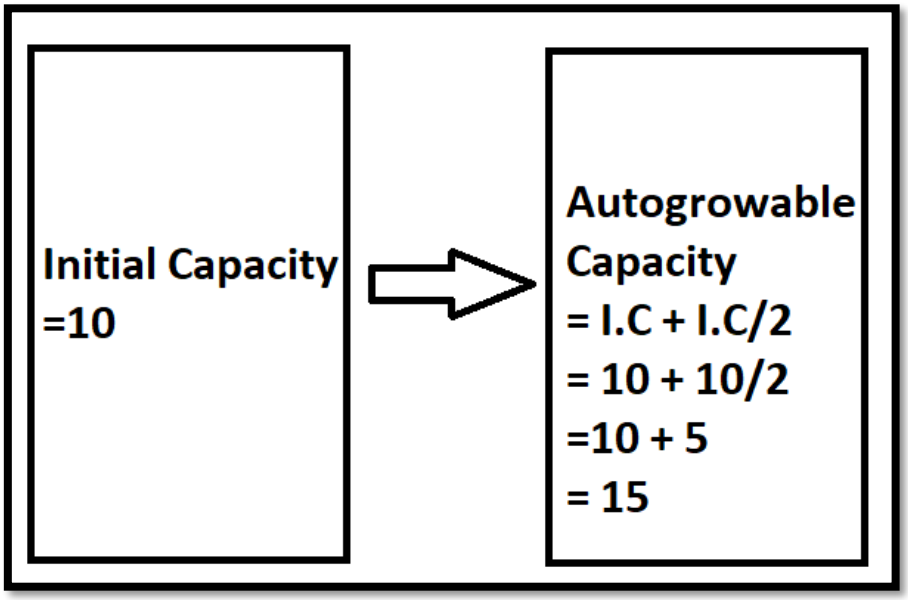
==> Insertion order is preserved.

==> It allows duplicate values.

==> It allows NULL values.

==> Initial capacity is 10, Autogrowable capacity is half of its initial capacity.

==> It is Homogeneous as well as Heterogeneous in Nature.




```
package collection_programs;

import java.util.*;

public class CollArrayList {

	public static void main(String[] args) {

		ArrayList al = new ArrayList();

		/*Adding Elements*/
		al.add(50);
		al.add(97);
		al.add("pushpa");
		al.add(24.56);
		al.add(50);
		al.add(null);

		/*Fetch all the Elements at a time*/
		System.out.println(al);
```

```
                /*Searching Elements*/
                System.out.println(al.contains("pushpa"));

                /*Fetching a Particular Element*/
                System.out.println(al.get(2));

                /*Deletion Operation*/
                System.out.println(al.remove(2));
                System.out.println(al);

                /*Modification of Particular Cell*/
                al.set(4,"Kattappa");
                System.out.println(al);

                /*Modification of Entire List*/
                al.add(1,"Bahubali");
                System.out.println(al);
        }
}
```

OUTPUT

```
[50, 97, pushpa, 24.56, 50, null]
true
pushpa
pushpa
[50, 97, 24.56, 50, null]
[50, 97, 24.56, 50, Kattappa]
[50, Bahubali, 97, 24.56, 50, Kattappa]
```

LINKEDLIST

==> It is introduced in the version **JDK1.2**.

==> Insertion order is preserved.

==> It is Homogeneous as well as Heterogeneous in nature.

==> It supports duplicate values.

==> It allows NULL values.

==> There is no initial capacity.

```java
package collection_programs;

import java.util.*;

public class CollLinkedList {

	public static void main(String[] args) {

		LinkedList ll = new LinkedList();

		/*Adding Elements*/
		ll.add(50);
		ll.add(97);
		ll.add("Rocky");
		ll.add(24.56);
		ll.add(50);
		ll.add(null);

		/*Fetch all the Elements at a time*/
		System.out.println(ll);

		/*Searching Elements*/
		System.out.println(ll.contains("Rocky"));

		/*Fetching a Particular Element*/
		System.out.println(ll.get(2));

		/*Deletion Operation*/
		System.out.println(ll.remove(2));
		System.out.println(ll);

		/*Modification of Particular Cell*/
		ll.set(4,"Adhira");
		System.out.println(ll);

		/*Modification of Entire List*/
		ll.add(1,"Allu Arjun");
		System.out.println(ll);

		/*To get the Length*/
		System.out.println("Length of the list is : "+ll.size());
```

```
            /*To check the list is empty or not*/
            System.out.println(ll.isEmpty());

            /*To clear out list*/
            ll.clear();
            System.out.println(ll.isEmpty());
        }
}
```

OUTPUT

```
[50, 97, Rocky, 24.56, 50, null]
true
Rocky
Rocky
[50, 97, 24.56, 50, null]
[50, 97, 24.56, 50, Adhira]
[50, Allu Arjun, 97, 24.56, 50, Adhira]
Length of the list is : 6
false
true
```

PRIORITY QUEUE

==> Priority Queue introduced in the version **JDK 1.5**.

==> It is Homogeneous in nature.

==> Duplicate values are allowed.

==> NULL values are not allowed. If we try to add NULL values we will get NULL Pointer Exception.

==> Initial capacity is 11.

==> It will increase by double of its initial capacity.

NOTE :

If we try to add mixed objects (or) different type of objects we will get Class Cast Exception

---

==> Insertion order is not preserved because priority queue follows Binary Structure.

RULES OF TREE STRUCTURE

==> One Parent node can have maximum two children and minimum ozero child.

==> Tree Structure flows top to bottom and left to right.

==> Parent node value should be lesser than Child node value.

```
package collection_programs;

import java.util.PriorityQueue;

public class CollPQueue {

	public static void main(String[] args) {

		PriorityQueue pq = new PriorityQueue();
		pq.add(30);
		pq.add(20);
		pq.add(100);
		pq.add(7);
		pq.add(67);
		System.out.println(pq);
		System.out.println(pq.peek());
		System.out.println(pq);
		System.out.println(pq.poll());
		System.out.println(pq);
		System.out.println(pq.poll());
		System.out.println(pq);
		System.out.println(pq.poll());
		System.out.println(pq);
		System.out.println(pq.poll());
		System.out.println(pq);
		System.out.println(pq.poll());
		System.out.println(pq);
		System.out.println(pq.remove());
		System.out.println(pq);
	}
}
```

OUTPUT

```
[7, 20, 100, 30, 67]
```

7
[7, 20, 100, 30, 67]
7
[20, 30, 100, 67]
20
[30, 67, 100]
30
[67, 100]
67
[100]
100
[]
NULL

DIFFERENCE BETWEEN PEEK( ) AND POLL( )

PEEK	POLL
Peek method can perform only one operation.	Poll method can perform two operations.
With the help of peek method we can get only the first element in Queue.	Poll method will get the first element from the queue and remove it from the queue.

DIFFERENCE BETWEEN POLL( ) AND REMOVE( )

POLL	REMOVE
After clearing queue if we use poll() method will get NULL method.	After clearing queue if we used remove() method we will get no Search Element Exception

HASHSET

==> HashSet is introduced in the version **JDK 1.2**.

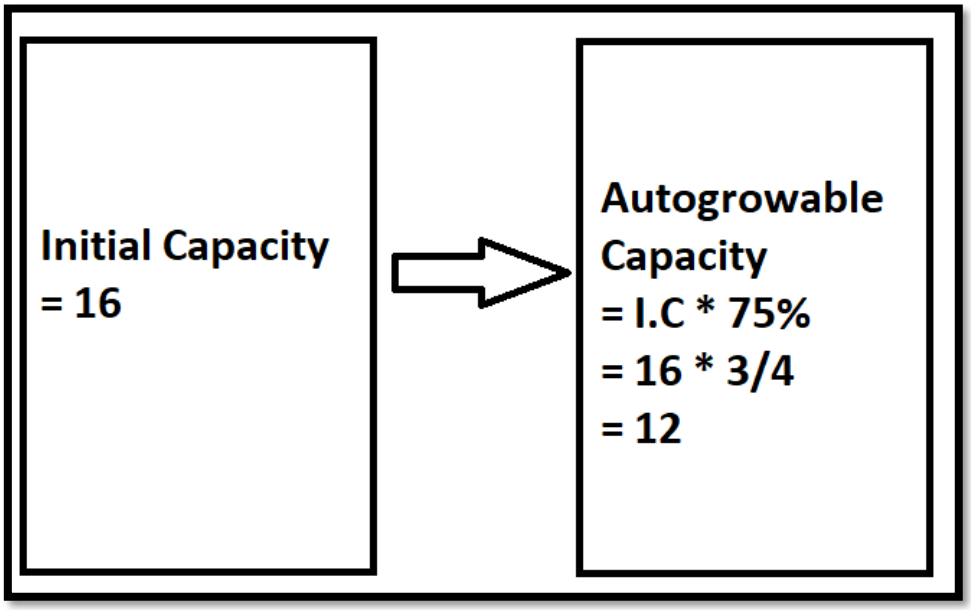
==> It is Homogeneous as well as Heterogeneous in nature.

==> NULL values are allowed.

==> Duplicate values are not allowed.

==> Insertion order is not preserved because it is following Hashing order.

==> Initial Capacity is 16. Fill ratio is 0.7 or 75%. It means whenever 75% of Initial Capacity will be fill. It will autogrowable by double of its initial capacity.




```
package collection_programs;

import java.util.HashSet;

public class CollHashSet {

	public static void main(String[] args) {

		HashSet hs = new HashSet();
		hs.add(30);
		hs.add(40);
		hs.add("HELLO");
		hs.add(30);
		hs.add(null);
		System.out.println(hs);
	}
}
```

OUTPUT

[null, HELLO, 40, 30]

LINKED HASHSET

==> Linked HashSet is introduced in **JDK 1.4** version.

==> It is Homogeneous and Heterogeneous in nature.

==> Duplicate values are not allowed.

==> NULL values are allowed.

==> Initial capacity is 16. If the fill ratio is 0.75 (or) 75%, it means whenever 75% of initial capacity will be filled it will grow by double of its Inital Capacity.

NOTE :

==> Linked HashSet is the combination of Linked List and HashSet.

==> Because of LinkedList insertion order is preserved.

==> Because of HashSet duplicate values are not allowed.

```
package collection_programs;

import java.util.LinkedHashSet;

public class CollLinkedHashSet {

	public static void main(String[] args) {

		LinkedHashSet lhs = new LinkedHashSet();
		lhs.add(30);
		lhs.add("HELLO");
		lhs.add(30);
		lhs.add(25.56);
		lhs.add(null);
		System.out.println(lhs);
	}
}
```

OUTPUT

[30, HELLO, 25.56, null]

TREESET

==> Introduced in **JDK1.2** version.

==> Insertion order is ascending order.

==> Duplicate values are not allowed.

==> NULL values are not allowed. If we try to add we will get NULL POINTER EXCEPTION.

==> It is Homogeneous in nature.If we try to add different objects we will get Class Cast Object.

```
package collection_programs;

import java.util.TreeSet;

public class CollTreeSet {

	public static void main(String[] args) {

		TreeSet ts = new TreeSet();
		ts.add(40);
		ts.add(40);
		ts.add(10);
		ts.add(100);
		ts.add(20);
		ts.add(5);
		System.out.println(ts);
	}
}
```

OUTPUT

[5, 10, 20, 40, 100]

CURSOR

==> In java, Cursor means Traversing.

==> Traversing means one element to another element.

==> We have three cursors

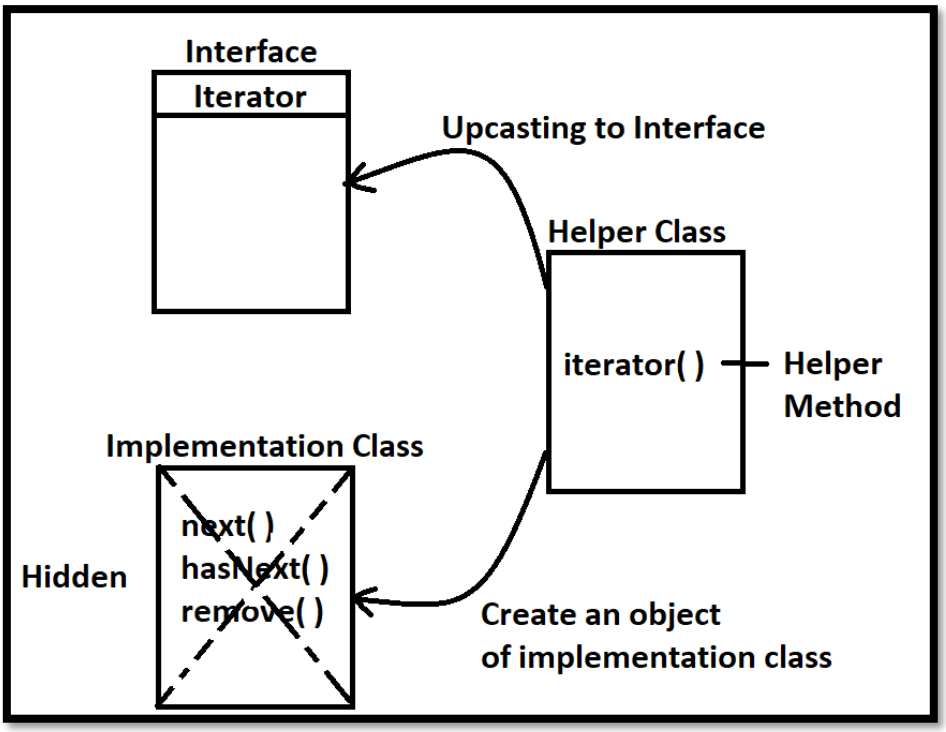
1. Iterator → Entire Collection
2. ListIterator → Only in List specific implementation
3. Enamuration → Legacy class

ITERATOR

==> Iterator is an interface present in **java.util** package.

==> Inside Iterator interface there are three methods

1. hasNext();
2. next();
3. remove();



```
package collection_programs;

import java.util.ArrayList;
import java.util.Iterator;

public class Coll_Iterator {

	public static void main(String[] args) {

		ArrayList al = new ArrayList();
		al.add(40);
		al.add(20);
		al.add(50);
		Iterator itr = al.iterator();
		System.out.println(itr.hasNext());// Searching Next Element
		System.out.println(itr.next()); // Print Present Element
		System.out.println(itr.hasNext());
		System.out.println(itr.next());
		System.out.println(itr.hasNext());
		System.out.println(itr.next());
	}
}
```

OUTPUT

```
true
40
true
20
true
50
```

LISTITERATOR

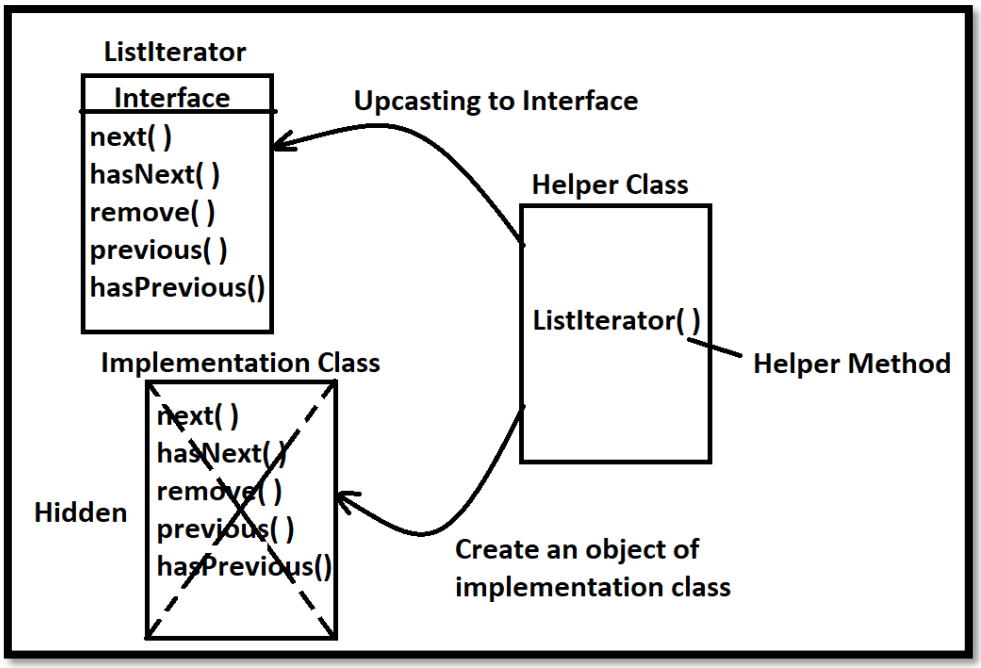
==> ListIterator is present in **java.util.package**.

==> There are five methods in ListIterator

1.) next( ) 2.) hasNext( )

3.) remove( )

4.) previous( ) 5.) hasPrevious( )



DIFFERENCE BETWEEN ITERATOR AND LISTITERATOR

ITERATOR	LISTITERATOR
Iterator is applicable in entire collection	ListIterator is applicable only in list specific implementation classes
Iterator has three methods	ListIterator has five methods
Iterator can perform only Forward Traversing	ListIterator can perform Forward Traversing as well as Backward Traversing

GENERIC

==> Generic is a concept which is introduced in the Java in the version **JDK1.5**, to make collection 100% type safe in nature.

==> We can declare generic inside the brackets("< >").

Ex: ArrayList<Integer> al = new ArrayList<Integer>();

In this example we are making an ArrayLsit type safe only for Integer object.

DIFFERENCE BETWEEN COLLECTION AND COLLECTIONS

COLLECTION	COLLECTIONS
Collection is an Interface.	Collections is an Class.
Collection has three Sub Interfaces 1. List 2. Queue 3. Set	Collections doesn't have any Sub Classes
Collection methods are Non-Static and Abstract in nature	Collections methods are Static and Concrete in nature

DIFFERENCE BETWEEN LIST,QUEUE AND SET

LIST	QUEUE	SET
List allows duplicate values.	Queue also allows duplicate values	Set doesn't allows duplicate values
Insertion order preserved for List	Queue follows Binary Tree Structure	1. HashSet following Hashing order 2. LinkedHashSet insertion order is preserved because of Linked List. 3. TreeSet is following ascending order
In List we can use all three cursors	In Queue we can use only Iterator	In Set we can ud=se only Iterator

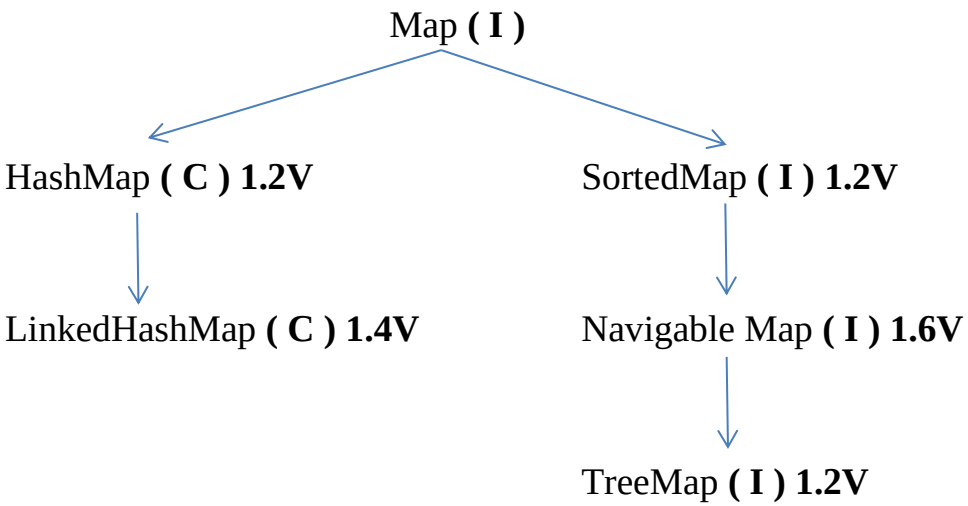
DIFFERENCE BETWEEN ARRAYLIST AND VECTOR

ARRAYLIST	VECTOR
ArrayList was introduced in the version **JDK 1.2**	Vector was introduced in the version **JDK 1.0**
ArrayList increases by half of its initial capacity	Vector increases by double of its initial capacity
In ArrayList we can use two cursor	In Vector we can use all cursors
ArrayList methods are Non-Thread safe	Vector methods are Thread safe
ArrayList methods are Non-Synchronized in nature	Vector methods are Synchronized in nature
ArrayList works faster than Vector	Vector works slower than ArrayList

DIFFERENCE BETWEEN ARRAYLIST AND LINKEDLIST

ARRAYLIST	LINKEDLIST
Initial Capacity is 10	Don't have Initial Capacity
ArrayList is the best choice to add (or) remove elements from beginning and ending	LinkedList is the worst choice to add (or) remove elements from beginning and ending
ArrayList is the worst choice to add (or) remove elements in the middle	LinkedList is the best choice to add (or) remove elements in the middle

HIERARCHY OF MAP



NOTE :

Maps always follow key and value pair. Values can be duplicate but Keys can't be duplicate

---

HASHMAP

==> It is introduced in **JDK 1.2** version.

==> Insertion order is not preserved. It is following hashing order.

==>Initial capacity is 16. Fill ratio is 0.75 and it will increase by double of its initial capacity.

==> Duplicate values are allowed but duplicate key are not allowed.

==> NULL values are allowed but NULL key are not allowed.

```
package collection_programs;

import java.util.HashMap;

public class CollHashMap {

	public static void main(String[] args) {

		HashMap hm = new HashMap();
		hm.put(2 , "Ram");
		hm.put(3 , "Raj");
		hm.put(5 , "Jay");
		hm.put(4 , "Kiran");
		System.out.println(hm);
	}
}
```

OUTPUT

```
{2=Ram, 3=Raj, 4=Kiran, 5=Jay}
```

LINKEDHASHMAP

==> It is introduced in the **JDK1.4** version.

==> Initial capacity is 16. Fill ratio is 0.75% and it will autogrowable by double of its initial capacity.

==> It is Homogeneous as well as Heterogeneous in nature.

==> Duplicate values are allowed. But duplicate key are not allowed.

==> Insertion order is preserved.

==> NULL values are allowed, but NULL keys are not allowed.

```
package collection_programs;

import java.util.LinkedHashMap;

public class CollLinkedHashMap {

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

```
            LinkedHashMap lhm = new LinkedHashMap();
            lhm.put(1 , "Ram");
            lhm.put(2 , "Raj");
            lhm.put(3 , "Jay");
            System.out.println(lhm);
        }
}
```

OUTPUT

{1=Ram, 2=Raj, 3=Jay}

TREEMAP

==> It is introduced in **JDK1.2** version.

==> Insertion order is ascending order.

==> No Initial capacity is there.

==> Don’t allow duplicate values.

==> NULL values are not allowed. If we try to add NULL values will get NULL Pointer Exception.

==> It is Homogeneous in nature. If we try to add mixed objects we will get Class Cast Exception.

```
package collection_programs;

import java.util.TreeMap;

public class CollTreeMap {

        public static void main(String[] args) {

            TreeMap tm = new TreeMap();
            tm.put(2 , "Ram");
            tm.put(3 , "Raj");
            tm.put(4 , "Jay");
            System.out.println(tm);
        }
}
```

OUTPUT

{2=Ram, 3=Raj, 4=Jay}

DIFFERENCE BETWEEN MAP AND COLLECTION

MAP	COLLECTION
To add elements in Map we need put( ) method	To add elements in Collection we need add( ) method
Map follows key and value pair	Collection doesn't follow key and value pair but it has elements
Cursor concept is not applicable in Map	Cursor is applicable in Collection

DIFFERENCE BETWEEN HASHSET AND LINKEDHASHSET

HASHSET	LINKEDHASHSET
HashSet version is **JDK 1.2**	LinkedHashSet version is **JDK 1.4**
In HashSet insertion order is not preserved	In LinkedHashSet insertion order is preserved because of Linked List

DIFFERENCE BETWEEN TREESET AND LINKEDHASHSET

TREESET	LINKEDHASHSET
TreeSet version is **JDK1.2**	LinkedHashSet version is **JDK 1.4**
TreeSet is Homogeneous in nature	LinkedHashSet is both Homogeneous as well as Heterogeneous in nature.
TreeSet doesn't allow NULL values	LInkedHashSet allow NULL values
TreeSet insertion order is ascending order	LinkedHashSet insertion order is preserved.
TreeSet doesn't have initial capacity	LinkedHashSet initial capacity is 16

THREAD

==> Thread is nothing but the path for execution. If we don't declare in a thread by default it will take main as thread.

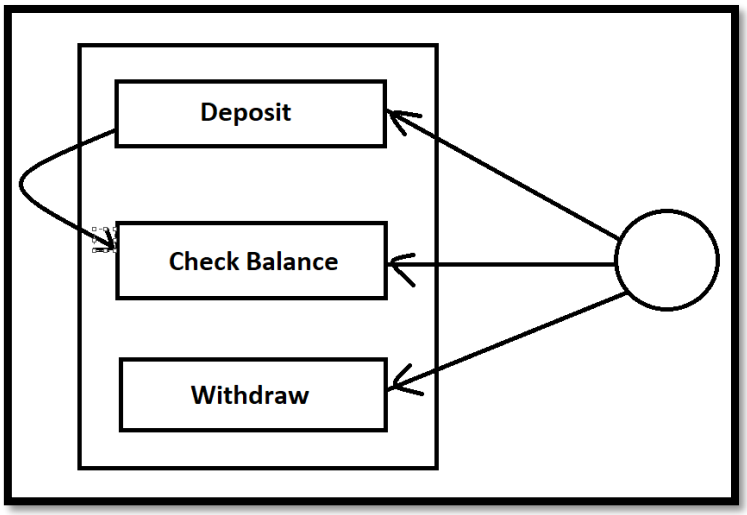
==> In **java.lang** package there is an inbuilt class called as thread is present.

MULTI-TASKING

==> Doing multiple task at a same time but not in an efficient manner to save time is called Multi-tasking.

MULTI-THREADING

==> Doing multiple task at a same time in a more efficient way is called Multi-threading.



Q. Can we create our own Thread?

A: Yes, we can create our own thread class.

Q. How many ways we can create our own Thread?

A: There are two ways to create our own thread

1. With the help of Thread Class
2. With the help of Runnable Interface

Steps to create thread With the help of Thread Class

1. Create our own class.

2. Extends the Thread class.
3. Over-ride run method.
4. Create Main method.
5. Create Object of own class.
6. Call start() to start the Thread.

```
package thread_programs;

public class With_TClass extends Thread {

	public void run()
	{
		System.out.println("Thread Started");
	}
	public static void main(String[] args)
	{
		With_TClass tc = new With_TClass();
		tc.start();
	}
}
```

OUTPUT

```
Thread Started
```

```
package thread_programs;

public class With_TClass extends Thread {

	public void run()
	{
		System.out.println("Thread Started");
	}
	public static void main(String[] args)
	{
		With_TClass tc = new With_TClass();
		tc.start();
		tc.start();
	}
}
```

OUTPUT

```
Thread Started
```

```
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.start(Unknown Source)
        at thread_programs.With_TClass.main(With_TClass.java:13)
```

NOTE

We cannot use 2 start() methods at a time. If we use we will get Exception.

Steps to create thread With the help of Runnable Interface

1. Create our own class.
2. Implements Runnable interface.
3. Over-ride run method.
4. Create Main method.
5. Create Object of own class.
6. Create Object of Thread class.
7. Pass own class object reference to the Thread class constant argument
8. Call start() to Start method.

```
package thread_programs;

public class With_RInterface extends Thread{

        public void run()
        {
                System.out.println("Thread Started");
        }
        public static void main(String[] args)
        {
                With_RInterface ri = new With_RInterface();
                Thread t1 = new Thread(ri);
                t1.start();
        }
}
```

OUTPUT

```
Thread Started
```

ITC METHODS

==> ITC means Inter Thread Communication.

==> As a developer we cannot control Thread but we can communicate with thread with the help of ITC methods.

==> we have 5 ITC methods.

1. wait()
2. wait(arg)
3. wait(arg1,arg2)
4. notify()
5. notifyAll()

STATIC LOCK

==> If thread is static and synchronized in nature after entering one thread inside stack area static pool area will be locked.

OBJECT LOCK

==> If thread is non-static and synchronized in nature after entering one thread inside stack area object area will be locked.

wait()

==> Whenever we are using the method the threads which are not executing will be waiting inside waiting pool area

---

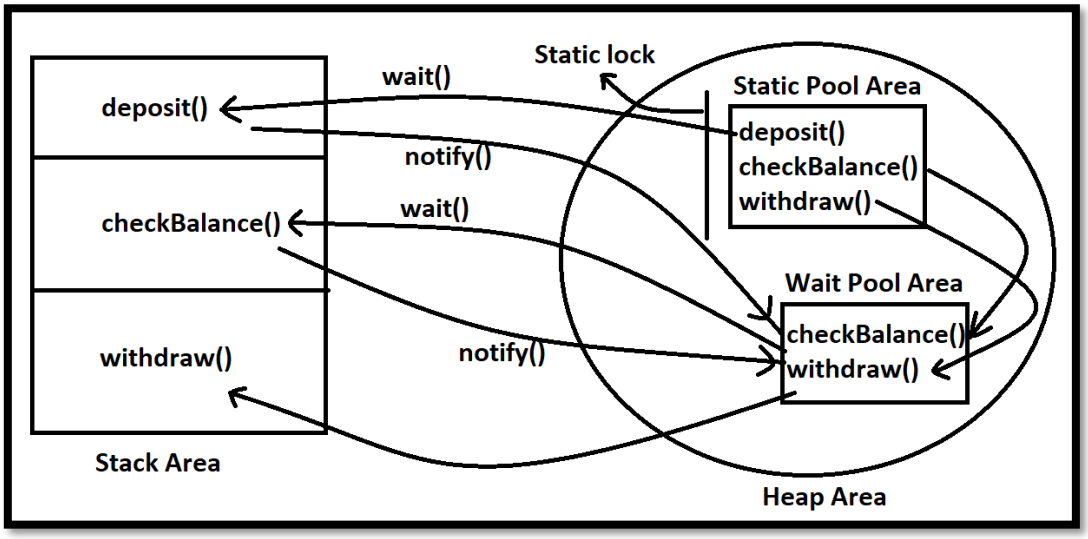
NOTE

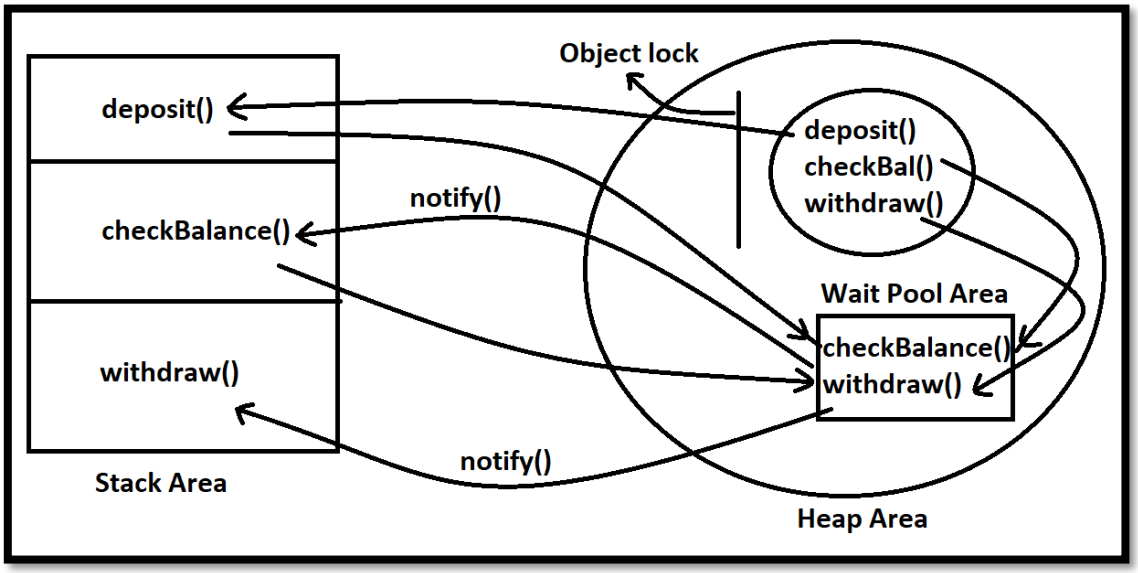
If we don't notify those threads it will be there for entire life time.This is the drawback of wait() method. To remove this drawback wait(arg) and wat(arg1,arg2) methods are introduced.

---

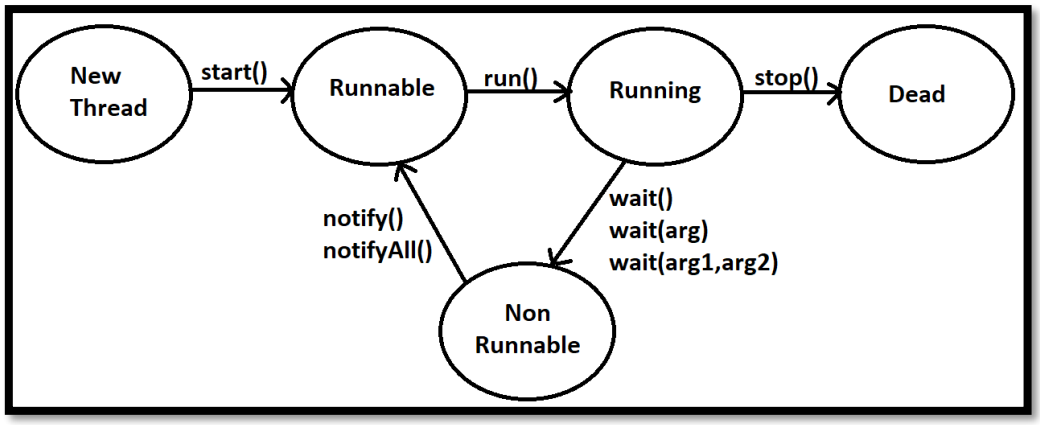
notifyAll()

==> With the help of notifyAll() method we can notify all the threads which are present in waiting pool area so that all the threads will try to execute at a same time. So the programmer (or) developer may not communicate with the threads properly.





THREAD LIFE CYCLE



STIRNG BUFFER	STRING BUILDER
String Buffer is Thread Safe in nature.	String Builder is Non Thread in nature.
String Buffer is Synchronized in nature.	String Builder is Non Synchronized in nature.
String Buffer works slowly	String Builder works fastly

STRING	STRING BUFFER AND BUILDER
String objects are Immutable	String Buffer and Builder objects are Mutable
We can create object in two ways	We can create object in one way **With new keyword**
To perform concatenation we are using concat() method	To perform concatenation we are using append() method

COMPARABLE	COMPARATOR
Inside Comparable one method is present called as compareTo()	Inside Comparator one method is present called as compare ()
With Comparable we can perform Default Natural Sorting Order(DNSO)	With Comparator we can perform Customized Sorting Order(DNSO)

JAVA BEAN CLASS

==> Java Beans class can be performed by using Private Members and Public Methods.

==> Public Methods are of 2 types.

1. Getters 2. Setters

==> Getters will help us to access private members to another class and we can fetch also. It is also called **“Accessors”**.

==> Setters are used as modification purpose. It is also called **“Modifiers”**.

Example:

```
package java_beans_programs;

public class Bank {

	private String accname;
	private long accnum;
	private String address;
	private int age;
	private long mobnum;

	public String getAccname()
	{
		return accname;

	}
	public void setAccname(String accname)
	{
		this.accname = accname;
	}

	public long getAccnum()
	{
		return accnum;
	}
	public void setAccnum(long accnum)
	{
		this.accnum = accnum;
	}

	public String getAddress()
	{
		return address;
	}
	public void setAddress(String address)
	{
```

```
		this.address = address;
	}

	public int getAge()
	{
		return age;
	}
	public void setAge(int age)
	{
		this.age = age;
	}

	public long getMobnum()
	{
		return mobnum;
	}
	public void setMobnum(long mobnum)
	{
		this.mobnum = mobnum;
	}
}
package java_beans_programs;

public class Bank_Details {

	public static void main(String[] args) {

		Bank b = new Bank();
		b.setAccname("Rakesh");
		b.setAccnum(123456789);
		b.setAddress("KPHB Colony");
		b.setAge(26);
		b.setMobnum(987654321L);

		String name = b.getAccname();
		long accountNumber = b.getAccnum();
		String address = b.getAddress();
		int age = b.getAge();
		long mobileNumber = b.getMobnum();
		System.out.println("Name is "+name);
		System.out.println("Acc Number is "+accountNumber);
		System.out.println("Address is "+address);
		System.out.println("Age is "+age);
```

```
            System.out.println("Mobile Number is "+mobileNumber);
        }
}
```

OUTPUT

```
Name is Rakesh
Acc Number is 123456789
Address is KPHB Colony
Age is 26
Mobile Number is 987654321
```

POJO CLASS

==> POJO means Plain Old Java Object

```
package java_beans_programs;

public class NewBank {

        private String accname;
        private long accnum;
        private String address;
        private int age;
        private long mobnum;

        public NewBank(String accname , long accnum , String address , int age,
long mobnum)
        {
                this.accname = accname;
                this.accnum = accnum;
                this.address = address;
                this.age = age;
                this.mobnum = mobnum;
        }

        public String getAddress()
        {
                return address;
        }
        public void setAddress(String address)
        {
                this.address = address;
```

```
	}

	public int getAge()
	{
		return age;
	}
	public void setAge(int age)
	{
		this.age = age;
	}

	public long getMobnum()
	{
		return mobnum;
	}
	public void setMobnum(long mobnum)
	{
		this.mobnum = mobnum;
	}

	public String getAccname()
	{
		return accname;
	}

	public long getAccnum()
	{
		return accnum;
	}
}



package java_beans_programs;

public class NewBank_Details {

	public static void main(String[] args) {

		NewBank nb = new NewBank("Ramu", 1234567890,
"KPHB/JNTU", 29, 987654321);

		System.out.println("Acc Name is "+ nb.getAccname());
```

```
            System.out.println("Acc Number is "+ nb.getAccnum());
            System.out.println("Address is "+ nb.getAddress());
            System.out.println("Age is "+ nb.getAge());
            System.out.println("Mobile Number is "+ nb.getMobnum());
        }
}
```

OUTPUT

```
Acc Name is Ramu
Acc Number is 1234567890
Address is KPHB/JNTU
Age is 29
Mobile Number is 987654321
```

FILE HANDLING

==> File is an inbuilt class present in java.io package.

mkdir( )

==> With the help of mkdir( ) we can create a folder.

createNewFile( )

==> With the help of createNewFile( ) we create a new file.

Creation of file class object

Syntax: **File f = new File("file_path");**

```
package file_handling_programs;

import java.io.File;
import java.io.IOException;

public class Creating_Methods {

    public static void main(String[] args) throws IOException {

        File f1 = new File("D:\\java\\magic");
        f1.mkdir();
        System.out.println("Folder Created Successfully");
```

```
            File f2 = new File("D:\\java\\magic\\sample");
            f2.mkdir();
            System.out.println("Nested Folder Created Successfully");

            File f3 = new File("D:\\java\\magic\\sample.txt");
            f3.createNewFile();
            System.out.println("File Created Successfully");
        }
}
```

OUTPUT

```
Folder Created Successfully
Nested Folder Created Successfully
File Created Successfully
```

getAbsolutePath( )

==> With the help of this method we can fetch the absolute or exact path of the folder.

exists()

==> With the help of exists( ) we can check that the Folder or File is present or not.

delete()

==> With the help of delete( ) we can delete the file or folder.

```
package file_handling_programs;

import java.io.File;

public class PathExistDelete_Methods {

    public static void main(String[] args) {

            File f = new File("D:\\java\\magicmoment");
            f.mkdir();
            System.out.println("Folder Created Successfully");

            if(f.exists())
            {
                String path = f.getAbsolutePath();
```

```
                System.out.println("Folder Path is :"+path);
            }
            if(f.exists())
            {
                f.delete();
                System.out.println("File Deleted Successfully");
            }
            else
            {
                System.out.println("File is not Present");
            }
            System.out.println(f.exists());
        }
}
```

OUTPUT

```
Folder Created Successfully
Folder Path is :D:\java\magicmoment
File Deleted Successfully
false
```