★ For-Each Loop / Enhanced for loop:

For Each Loop is used to traverse a group of objects from an array / collection

Syntax:

```
for ( Datatype to be traversed   traversing Variable   • Array
                                                       • Collection )
{
}
```

) Package day3;

```
Class forEachLoop Example
{
  Psvm (---)
  {
   int[] a = {10, 20, 30, 40};
  for(int i=0; i<a.length; i++)
   {  sop[a[i]);
   }
   for [int i=a)
    {  sop(i);
    }
  Sop ("-----");
  double [] marks = {12.3, 56.1};
   for (double x : marks)
    {  Sop (x);
    }
  Sop ("-----");
   String [] fruits = {"Apple", "Banana", "Grapes", "Mango"};
        for (String f : fruits)
        {  sop (f);
        }
  } }
```

O/P

```
10
20
30
40
---
10
20
30
40
---
12.3
56.1
---
Apple
Banana
Grapes
Mango
```

* Datatypes:-

It is used to specify / indicate the type of data stored into a variable

- Datatypes are categorized into i) Primitive
  ii) Non-Primitive

1) Primitive Datatype: If a keyword is behaving as a Datatype we refer it as Primitive Datatype.

- We have 8 Primitive datatypes in Java & all have fixed size

=> Byte, long, float, char
Short, int, double, boolean

2) Non-Primitive Datatype:- If a class is behaving as a datatype, we refer it as Non-Primitive datatype.

- We can have 'n' no of non-primitive datatypes & doe not have fixed size
- The default value of all non-primitive datatypes are class type is null
  Ex: String class

* Note: The Collection framework does not support Primitive datatypes
∴ The alternative we is referred as wrapper class.

* Wrapper Class:-

The non-primitive version of primitive datatypes is called as "wrapper class"

- All wrapper classes are present in java.lang package ~~(String)~~
- toString() is overridden, in all wrapper classes & String class

Primitive Datatypes			Wrapper Class
1) byte (1 byte)	->	->	Byte
2) Short (2 bytes)	->	->	Short
3) Long (8 bytes)	->	->	Long
4) int (4 bytes)	->	->	Integer
5) float (4 bytes)	->	->	Float
6) double (8 bytes)	->	->	Double
7) boolean (1 bit)	->	->	Boolean
8) char (2 bytes)	->	->	Character

2) Package days

```
Public class Demo
{
  Psvm(--)
  {
    // Primitive Representation of 10
      int a=10;
    // Non-Primitive Representation of 10
     Integer b= new Integer(10);
      Sop(a+" "+b);
     Sop("------");
    // Primitive Representation of 'A'
      char x='A';
    // Non Primitive Representation of 'A'
      Character y= new Character('A');
      Sop(x+" "+y);
  }
}
```

* Auto boxing:- The process of Converting Primitive type to non-Primitive type is called Autoboxing.

* Auto unboxing:- The process of Converting non Primitive type to Primitive type is called Autounboxing

(2) class BoxingModel

```
{ psvm(--)
{
  // Auto-Boxing
  int a=10;
  Integer b=new Integer(a);
  Sop(a+" "+b);
  double i=3.4;
  Double j= new Double(i);
  Sop(i+" "+j);
Sop("------");
  // Auto-unboxing
Character x= new Character('z');
  char y=x;
  Sop(x+" "+y);
Integer c=new Integer(25);
  int d=c;
  Sop(c+" "+d);
}
}
```

o/p	10	10
	3.4	3.4
	z	z
	25	25

* Generics:

Is used to specify the element type of a collection Generics was introduced from JDK 1.5,

Syntax : < ElementType >

```
-> import java.util.*;
   class Example
   { psvm(--)
    { ArrayList<String> al = new ArrayList();
      al.add("Python");
      al.add("JavaScript");
      al.add("Java");
      for(String s: al)
       { sop(s);
       }
      Sop("---");
     LinkedList<Integer> ll = new LinkedList<Integer>();
      ll.add(10);
      ll.add(20);
      ll.add(30);
       for(int i: ll)
     { sop(i);
     }
    Sop("...");
     ArrayList<Character> l = new ArrayList<Character>();
        l.add('t');
        l.add('o');
        l.add('m');
         for(Character x: l)
        {
          sop(x);
```

```
Sop ("_____");
LinkedList l1 = new LinkedList();
    l1.add(12);
    l1.add(12.3);
    l1.add("Java");
 for (Object obj : l1 )
   {
     Sop(obj);
   }
 }
}
```

o/p: Python
JavaScript
Java

10
20
30
4
0
m

12
12.3
Java

---

5)
```
import java.util.ArrayList;
 class Test
{
 Psvm (--)
 {
  int a = 10;
  double b=12.5;
  ArrayList l= new ArrayList();
   l.add(a); // l.add (new Integer(10)); implicitly
   l.add(b); // l.add (new Double(12.5));
   l.add('z'); // l.add (new Character('z'));

  for (Object obj :l) //upCasting => Generalization
     { sop (obj);
     }
}}}
```

o/p: 10
12.5
a

// Object obj : [new Integer(10), new Double(12.5), new Character('z')]
// Superclass Reference and Subclass objects.

-> Vector?: Vector is a Pre-defined class present in java.util Package

-> Vector is also refered as a Legacy Collection as it was introduced from JDK 1.0.

-> The initial Capacity of Vector 10.

-> The incremental Capacity of Vector (Current Capacity * 2)

-> The underlined dataStructure of Vector is resizeable array or growable array.

-> Vector & arrayList are both Singular but the major difference is Vector is Thread Safe i.e., Synchronized where as Array_list is Non-Thread Safe i.e., Not Synchronized.

---

* Internal Working of Vector:-

(1) When we Create an Object of Vector an new array is Created based on Initial Capacity (or) Custom Capacity.

(2) If we add an object into ~~one the~~ Vector once the Vector is full the new array will be Created based on Incremental Capacity formula

(3) All the objects from old ~~with~~ array will be Copied to the newly Created Array.

(4) The reference Variable Pointing to the old Array gets de-reference & Starts Pointing to the new Created array

=> Difference Btw Vector & ArrayList:

Vector	ArrayList
① Introduced from JDK 1.0	① Introduced from JDK 1.2
② ~~Initial Capa~~ incremental Capacity (currentCapacity * 2)	② incremental Capacity (currentCapacity * 3)/2 + 1
③ Thread Safe (Synchronized)	③ Not-Thread Safe (Non Synchronized)
④ 4 Constructors	④ 3 constructors.

-> Package day4;

```
import java.util.Vector;
class Demo{
    psvm(){
    Vector v=new Vector();
    v.add(10);
    v.add(1.3);
    v.add(true);
    for(Object obj:v){
        sop(obj);
    }
}
}
```

O/P
10
1.3
true

Constructors in ArrayList, LinkedList and Vector:-

ArrayList
1. ArrayList()
2. ArrayList(int initialCapacity)
3. ArrayList(Collection c)

LinkedList
1. LinkedList()
2. LinkedList(Collection c)

Vector
1. Vector()
2. Vector(int initialCapacity)
3. Vector(int initialCapacity, int incrementalCapacity)
4. Vector(Collection c).

=> Package day 4;

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Vector;

class Example {
    psvm ( ) {
    // Internally an array gets created of default initial Capacity -> 10
     ArrayList a = new ArrayList ( );

    // Internally an array gets created of Custom initial Capacity -> 50
     ArrayList b = new ArrayList (50);

    // Objects will be Stored in the form of nodes
     LinkedList c = new LinkedList ( );

    // Internally an array gets created of default initial Capacity -> 10
     Vector d = new Vector ( );

    // Internally an array gets created of Custom initial Capacity -> 20
     Vector e = new Vector (20);

    // Internally an array gets created initial Capacity -> 20
    // Custom incremental Capacity = Current Capacity * 4
     Vector f = new Vector (20, 4); // CC*4

     ArrayList x = new ArrayList ( );
      x.add (10);

    // Adding One Collection into another
     LinkedList y = new LinkedList (x);
     y.add (20);

     ArrayList z = new ArrayList (2);
     z.add (40);
```

```
Sop(x); // [10]
Sop(y); // [10,20]
Sop(z); // [10,20,30]
Sop(l); // [10,20,30,40]
}
}
```

-> Package day11;

```
Class Student {
  int age;
  String name;

Student(int age, String name){
    this.age=age;
    this.name=name;
}
  @Override
Public String toString(){
  return "Age:" +this.age+ "Name:" +this.name;
}
}
import java.util.ArrayList;
Class Solution{
  Psvm(){
  Student S1=new Student(21,"Tom");
  Student S2=new Student(23,"John");
  Student S3=new Student(22,"Paul");
ArrayList<Student> l=new ArrayList<Student>();
  l.add(S1);
  l.add(S2);
  l.add(S3); // l.add(new Student(22,"Paul"));
  // if toString() is Overridden
  for(Student S:l)
    Sop(S);
```

```
Sop("-----");
// if toString() is not Overridden
for(Student S:l)
  Sop("Age:"+S.age+"Name:"+S.name);
}
}
}
```

```
O/P
Age:21 name:Tom
Age:23 name:John
Age:22 name:Paul
```

What is Truncation in Java?

a) Floating-point value assigned to a float type

b) Floating

String String:

-> String is a Predefined final class present in java.lang package

-> Since String is a final class we cannot inherit String class.

-> String objects are immutable in nature.

-> String can also be referred as a non-primitive Datatype bcs String is class & default value for String (or) any class type is always null

-> String can also be referred as a set of sequence of characters

-> String objects can be created in two ways.

(1) As a literal (without using new operator)

(2) Using new Operator.

Example

(1) String s = "Guldu";

(2) String s = new String("Dinga");

-> String objects will get stored inside a memory location called as String Pool.

-> String pool is further divided into 2 parts

(1) Constant Pool.

(2) non-constant Pool.

-> String objects Created as literal will get Stored inside Constant Pool. & Constant Pool doesn't allow duplicates.

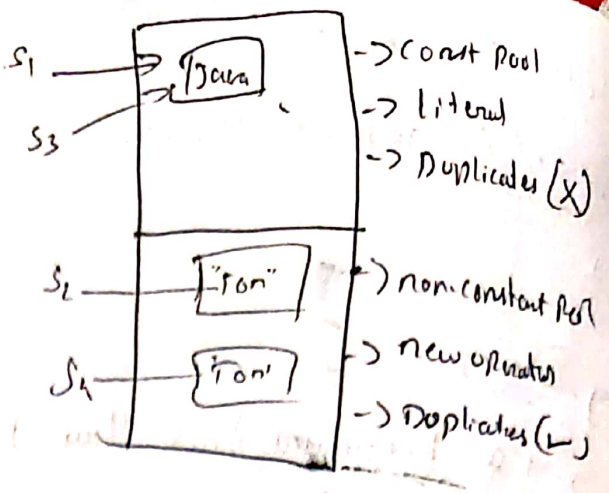
-> String objects Created using new operator will get stored inside nonConstant Pool & nonConstant Pool allows duplicates.

① String s1 = "Java";

② String s2 = new String("Jon");

③ String s3 = "Java";

④ String s4 = new String("Jon");



Note: String class also implicitly inherits Object class & has customatically overridden 3 method. from the object class

① hashCode()
② toString()
③ Equals()

-> toString() was supported to return the default String Representation of an object, but in String class toString method is overridden to return a sequence of Characters passed to the Constructor.

-> hashCode() was supported to return a unique random number for an object but in String class hashcode() is overridden to return the ascii value of the object.

-> Equals method was supported to Compare address of the 2 Objects, but in String class it is overridden to Compare the Content of 2 object

---

```
Eg: Class Employee {
    p sum (String[] args) {
    Employee emp = new Employee();
    Sop (emp);
    Sop (emp.toString());
    Sop ("-----");
    Sop (emp.hashCode());
    Sop ("-----");
    Employee e1 = new Employee();
    Employee e2 = new Employee();
    Sop (e1 == e2);
} } Sop (e1.equal(e2));
```

o/p

```
Com.Employee@...
Com.Employee@...
-----
2018699554
-----
false
false
```

```
class Example {
    Public Static kroid Main ( ){
      String S = new String ("a");
        Sop (S);
        Sop (S.toString());
        Sop ("....");
        Sop (S.hashCode());
        Sop ("....");
       String S1 = new String ("tom");
       String S2 = new String ("tom");
        Sop (S1 == S2);
        Sop (S1.Equals (S2));
    }}
```

```
o/p
a
a
---
97
---
false
true
---
```

note: String class implements the Bellowing interfaces

1. Serializable,
2. Comparable.
3. CharSequence.

-> toString(), hashCode(), Equals() are Pre-defined Examples of Overridden methods in String class

-> wait() in object class, Print() & Println() in Print Screen class are Pre-defined Examples for overloaded methods.

```
->class Test {
    PSvm ( )
    // Empty Representation of String
    String S1 = new String(); // String S1 = "";
     System.out.Println (S1);
      Sop ("....");
    // Passing a Set of Characters to Constructor
      String S2 = new String ("Tom");
        Sop (S2);
      Sop ("....");
    // Converting Characters Array Into String Object
      Char [] ch = {'d', 'j', 'n', 'g', 'i'};
      String S3 = new String(ch); Sop(S3);
```

```
o/p
---
Tom
---
cling.
```

```
class Employee {
    int id;
    String name;

    Employee (int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString () {
        return "Emp Id of " + this.name + " is " + this.id;
    }
}
```

o/p

```
Employee Id of Smith is 101
Employee Id of King is 201
Employee Id of Dinga is 301
```

```
import java.util.LinkedList;

class mainclass {
    public static void main ( ) {
        Employee e1 = new Employee (101, "Smith");
        Employee e2 = new Employee (201, "King");

        LinkedList <Employee> l = new LinkedList <Employee> ();
        l.add (e1);
        l.add (e2);
        l.add (new Employee (301, "Dinga"));
        for (Employee emp : l) {
            sop (emp);
        }
    }
}
```

→ write a Java Program to store 3 mobile objects into arraylist & traverse it using foreach loop after overriding toString() where in each mobile object will have 3 attributes i.e., Brand, modelno & cost

→ class Mobile {

String brand;
int modelno;
double cost;

Mobile (String brand, int modelno, double cost) {
this.brand = brand;
this.modelno = modelno;
this.cost = cost;
}

@Override
~~Public String toString()~~
Public String toString() {
return "BRAND:" + this.brand + "Modelno" + this.modelno + "cost:" + this.cost;
}
}

O/P
BRAND: mi Modelno: 101 Cost: 40000
BRAND: NOKIA modelno: 777 cost: 50000
BRAND: SAMSUNG Modelno: 619 Cost: 65000.0

import java.util.ArrayList;
class Mainclass {
public static void main(String[] args) {
Mobile m1 = new Mobile ("mi", 101, 40000);
Mobile m2 = new Mobile ("NOKIA", 0777, 50000);
Mobile m3 = new Mobile ("SAMSUNG", 619, 65000);
ArrayList<Mobile> al = new ArrayList<Mobile>();
al.add(m1);
al.add(m2);
al.add(m3);
for (Mobile m : al) {
System.out.println(m);
}
}
}

* Difference b/w Collection & Collections :-

-> Collection is a Pre-defined Interface Present in java.util Package, introduced from JDK 1.2.

-> Collections is a Pre-defined Class Present in java.util Package & introduced from JDK 1.2

Sort() :- Sort() is used to Sort a list of Homogeneous Objects.

Syntax :- Sort (List l)

Sort (List l, Comparator c)

-> import java.util.ArrayList;
import java.util.Collections;

```
class Demo{
    psvm ( ) {
    ArrayList <Integer> l = new ArrayList<Integer>();
     l.add (20);
     l.add (30);
     l.add (10);
    Sop ("Before Sorting:");
    for (int i : l) {
       Sop (i);
    }
     Collections.Sort (l);
    Sop ("After Sorting:");
    for (int i : l) {
       Sop (i);
}}}
```

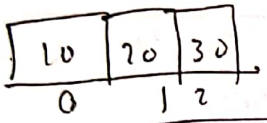
o/p :

```
Before Sorting:
20
30
10
After Sorting:
10
20
30
```

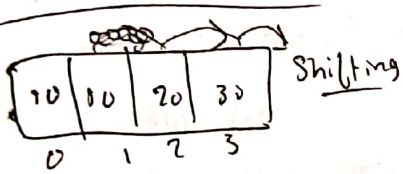
note: we Prefer LinkedList over ArrayList Whenever there is lot of insertion & deletion in b/w because Shifting of objects is involved in array List which reduces the efficiency, where as in LinkedList we don't have Shift operation

=> we Prefer ArrayList over LinkedList when we want to Store the objects and retrive those objects without any manipulation bcs ArrayList is Sequential in Nature where in traversing is much faster Compared to LinkedList

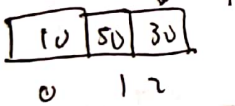
ArrayList



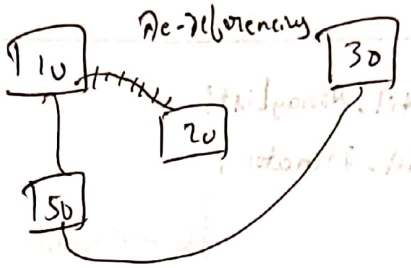
-> Add 50 in index-1



=> Remove from index-2



LinkedList



=> Iterator: Iterator is a Pre-defined interface Present in java.util Package

-> Iterator was introduced from JDK 1.2

-> Iterator is use to traverse a Group of objects from a collection

-> Iterator is Unidirectional.

Important Methods Under Iterator:

iterator(): this method is use to generate the iterable objects address.

Syntax: Public Iterator iterator()

2. hasNext():

-> This Method is use to check if there is an object present in the next position or not.

Syntax:- Public boolean hasNext()

3. next():

-> This Method is use to return the object one by one

Syntax:- Public Object next()

```
-> import java.util.ArrayList;
   import java.util.Iterator;

   class Test {
      psvm () {

   ArrayList<Integer> l = new ArrayList<Integer>();
      l.add(10);
      l.add(20);
      l.add(30);

      Iterator<Integer> i = l.iterator();
      while (i.hasNext()) {
         Sop (i.next());
      }
      Sop ("----");

      for (int x : l) {
         Sop (x);
      }
      Sop ("----");
      for (int j=0; j<l.size(); j++) {
         Sop (l.get(j));
      }
   }
}
```

o/p

```
10
20
30
----
10
20
30
----
10
20
30
----
```

⇒ write a Java Program to Store heterogeneous Objects into LinkedList & traverse it using Iterator.

⇒ import java.util.~~ArrayList~~ LinkedList;
import java.util.Iterator;

~~class Main {~~
~~public static void main (String[] args) {~~
LinkedList~~<String>~~ l = new ~~ArrayList~~ LinkedList();
l.add(

class Test {
int cost;
String name;
Double Salary;
Test (int cost, String name, Double Salary) {
this.cost = cost;
this.name = name;
this.Salary = Salary;
}

class main {

(2)

```
-> import java.util.ArrayList;
   import java.util.Iterator;

Class Example {
      PS vm () {
ArrayList<String> l = new ArrayList<String>();
   l.add ("java");
   l.add ("web");
   l.add ("sql");

Iterator<String> i = l.iterator();
while (i.hasNext()) {
     Sop (i.next());
   }
}
```

old data
web
SQL

Iterator — Pre defined interface, java util, JDK 1.2 — Uni-Directional
- iterator()
- hasNext()
- next()

ListIterator — .pdf, java.util, JDK 1.2 — Bi-Directional
- List/ListIterator()
- hasPrevious()
- previous()

SET:

-> Set is a Pre-defined interface Present in java.util Package

-> Set was introduced from JDK 1.2.

-> The implementation classes of the Set interface are as follows.

1. HashSet();
2. LinkedHashSet();
3. TreeSet();

```
Set s = new HashSet();
        new LinkedHashSet();
        new TreeSet();
```

Specifications of Set interface:

1. Insertion order is not maintained
2. Duplication is not allowed.
3. Set is not index based.
4. Null values can be inserted.

Note: We cannot use get() on Set interface because get() is used to return the object based on indexed position but Set is not indexed based.
∴ In order to traverse the objects from a Set we make use of forEach Loop (or) Iterators

LIST	SET
1. LIST maintain insertion order	1. where as SET doesn't maintain insertion order
2. Duplication is allowed	2. Duplication is not allowed.
3. List is indexbased	3. Set is not indexed based

HashSet: HashSet is a pre-defined class present in java.util package

-> HashSet was introduced from JDK 1.2
-> The initial capacity of HashSet is 16
-> The load factor / fill ratio of HashSet is 0.75 (or) 75%

The underlined DataStructure of HashSet is HashTable

HashTable

key	value
hashcode of the object	object

-> import java.util.HashSet;

```
class Demo{
    psvm(){
    HashSet h = new HashSet();
      h.add(10);
      h.add(10);
      h.add(1.2);
      h.add(null);
      h.add("java");
    Sop("Size:"+h.size());
    Sop("-----");
    for(Object obj:h){
        Sop(obj);
    }
}
}
```

o/p

Size: 4

1.2
null
Java
10

Constructors present in HashSet:-

1. HashSet()
2. HashSet(int initialCapacity)
3. HashSet(int initialCapacity, float fillRatio)
4. HashSet(Collection c)

LinkedHashSet:-

-> LinkedHashSet is a pre-defined class present in java.util package
-> It was introduced from JDK 1.4
-> HashSet & LinkedHashSet are both similar Collections but the major difference is HashSet doesn't maintain insertion order whereas LinkedHashSet maintains insertion order.

```
import java.util.Iterator;
import java.util.LinkedHashSet;

class Test {
    psvm () {
LinkedHashSet<Integer> lhs = new LinkedHashSet<Integer>();
        lhs.add(10);
        lhs.add(20);
        lhs.add(15);

    for(Integer x : lhs) {
        sop(x);
    }
    sop("******");
    Iterator<Integer> i = lhs.iterator();
    while(i.hasNext()) {
        sop(i.next());
    }
  }
}
```

o/p
10
20
15

10
20
15

How SET Maintain Uniqueness (or) How SET doesn't allow duplicates?

Internally HashCode() & Equals() are Overridden there fore they are responsible for maintaining the Uniqueness in SET Interface.

TreeSet: TreeSet is a Predefined class present in java.util Package

-> TreeSet was introduced from JDK 1.2

-> TreeSet always Sorts the objects in Natural Sorting Order. i.e. (Ascending order)

-> import java.util.TreeSet;

```
Class Solution {
    Psvm () {
    TreeSet<Integer> t = new TreeSet<Integer>();
      t.add (30);
      t.add (40);
      t.add (20);
      t.add (30);
      t.add (80);
      t.add (10);

    for (int i : t) {
       Sop (i);
    }
}
```

O/P
10
20
30
40
80

Note: When we add an object into TreeSet internally CompareTo() gets called.

-> The return type of CompareTo() is int.

-> The CompareTo() returns (+1 when it is >) (-1 when it is <)
(0 when it is ==).

-> The Underlined datastructure of TreeSet is BinaryTree

-> The Traversing order of TreeSet is

- Left Node
- Parent Node
- Right Node

① t.add (30);

② t.add (40);

40.compareTo(30)

40 > 30 | 40-30
+1 | +10
RHS

③ t.add (20);

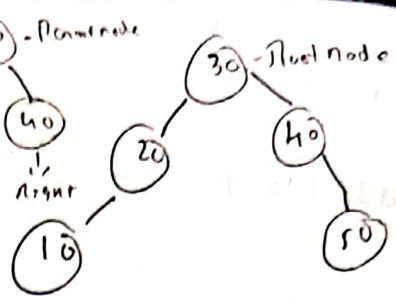
20.compareTo (30);

20 < 30 | 20-30
-1 | -10
LHS

④ t.add (30);

30.compareTo (30);

30 == 30 | 30-30
0 | 0
Dislcond



⑤ t.add (50);

50.compareTo(30);

50 > 30 | 50-30
+1 | +20
RHS

50.compareTo(40)

50 > 40 | 50-40
+1 | +10
RHS

⑥ t.add (10);

10.compareTo(50);

10 < 30 | 10-30
-1 | -20 => LHS

10.compareTo(20);

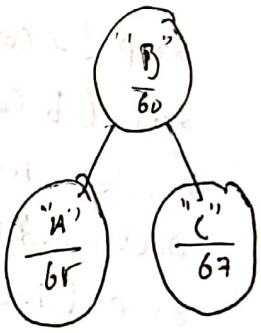
10 < 20 | 10-20
-1 | -10
LHS

Note: TreeSet will always allow the Heterogenous objects

```
=> import java.util.TreeSet;

class Solution {
    Psvm () {
    TreeSet <String> t = new TreeSet <String> ();
        t.add ("B");
        t.add ("C");
        t.add ("A");

    for (String i : t) {
        Sop (i);
    }
}
```



① t.add ("B")

② t.add ("C")

"C".compareTo("B");

67 | 66

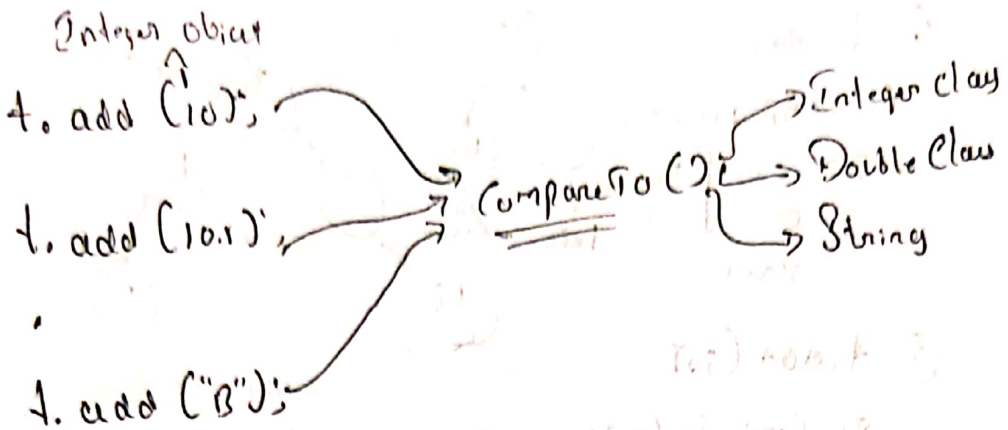
67 > 66 | 67-66
#1 | +1
RHS

③ t.add ("A")

"A".compareTo("B");

65 | 66

65-66 | 65-66
-1 | +1
LHS



-> Package day1:

```
class Mainclass
  PSVM (){
    String x="A";
    String y="B";
    Sop (x.compareTo(y)); //-1
    Sop (x.compareTo(x)); //0
    Sop (y.compareTo(x)); //1
    Integer a=20;
    Integer b=20;
    Sop (a.compareTo(b)); //-1
    Sop (a.compareTo(a)); //0
    Sop (b.compareTo(a)) //1
    Double i=4.5;
    Double j=3.4;
    Sop (i.compareTo(j)); //1
    Sop (j.compareTo(i)); //-1
    Sop (i.compareTo(i)); //0
}
```

Comparable:

-> Comparable is a pre-defined Interface present in java.lang Package
-> Comparable was introduced from JDK 1.2
-> Comparable is use to Compare the objects & Sort them Either in Ascending order (or) Descending order.
-> Comparable is use for default Sorting & natural Comparision
-> Comparable as an abstract Method called CompareTo()

Syntax: Public int CompareTo (E object c obj)
↓
Already Existing object

The return type of CompareTo() is int which returns (+1 when it is >) (-1 when it is <) (0 when it is ==).

Rules for implementing Comparable Interface

-> Class as to implement Comparable Interface
-> Specify Generics to indicate the type of objects to be compared
-> Override CompareTo() by specifing the business logic of Comparision

---

-> Package DefaultSorting;

```
Class Student implements Comparable<Student>{
    int age;
    Student(int age){
        this.age = age;
    }
    @Override
    Public String toString(){
        Return "Age:" + this.age;
    }
```

```
@Override
Public int compareTo(Student s) {
    return this.age - s.age; // ObjectToBe Entered Age - Already Existing object Age
}
}

Class SortStudent{
    Psvm (){
    Student S1: new Student (22);
    Student S2 = new Student (23);
    Student S3 = new Student (21);

  TreeSet<Student> t = new TreeSet<Student>();
    t.add (S1);
    t.add (S2);
    t.add (S3);

    for(Student Stel : t) {
      Sop (stel);
    }
  }
}
```

O/P

Age: 21
Age: 22
Age: 23

Tracing:

① t.add (S1);

② t.add (S2);

S2.compareTo(S1);

this → RHS

this.age - S.age => 23-22 => 1

③ t.add (S3);

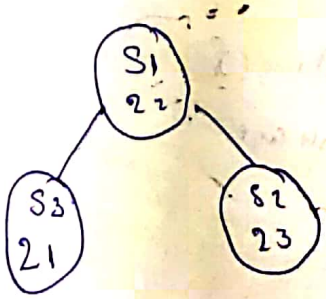
S3.compareTo(S1);

this → RHS

this.age - S.age => 21-22 (-1)

```
@Override
Public int compareTo(Student s)
{
} return this.age - s.age
```

Already Existing object



S1 Age-23

S2 Age-22

S3 Age-21

-> WAP to compare the Employee objects based on id & sort them in descending order.

default Package defaultSorting

```
class Employee implements Comparable<Employee>{
    int id;

Employee (int id){
    this.id = id;
}
@Override
Public String toString (){
    return "Id is " + this.id;
}
@Override
Dublic int compareTo (Employee e){
    return e.id - this.id;
}}
```

o/p
Id is 700
Id is 400
Id is 200

-> import java.util.TreeSet;

```
class SortEmployee{
    Psvm {
    Employee e1 = new Employee(200);
    Employee e2 = new Employee(400);
    Employee e3 = new Employee(700);

TreeSet<Employee> t = new TreeSet<Employee>();
        t.add(e1);
        t.add(e2);
        t.add(e3);
for (Employee emp : t);  SOP(emp);
```

Note: All Wrapper Classes & String Class implements the Comparable interface & has Automatically Overridden CompareTo()

-> Class Person implements Comparable<Person>{

```
int age;
String name;
Double height;
Person (int age, String name, double height){
    this.age = age;
    this.name = name;
    this.height = height;
}
@Override
Public String toString(){
  return "Person [age=" + age + ", name=" + name + ", height=" + height + "]";
}
@Override
Public int CompareTo(Person p){
  return this.name.compareTo(p.name);
}}
```

// return this.age - p.age; -> Comparing based on age in Ascending order
// return p.age - this.age; -> " " " " " Descending order
// return this.name.compareTo(p.name); -> " " " name in Ascending order
// return p.name.compareTo(this.name); -> " " " " " Descending order
// return this.height.compareTo(p.height); -> " " " height in Ascending order
// return p.height.compareTo(this.height); -> " " " " " Descending order

```
import java.util.TreeSet;
class SortPerson {
PSVM ( ) {
Person P1 = new Person (25, "B", 5.6);
Person P2 = new Person (21, "C", 5.5);
Person P3 = new Person (26, "A", 5.4);

TreeSet <Person> t = new TreeSet <Person>();
   t.add (P1);
   t.add (P2);
   t.add (P3);

  for (Person P: t) {
     SOP (P);
  }
}
}
```

o/p

Person [age: 25, name = A, height = 5.4]

Person [age = 25, name: B, height: 5.6]

Person [age = 21, name = C, height = 5.5]

@Sorting

① t.add (P1);

② t.add (P2);

P2.compareTo(P1)

↓ this — P

this.name.compareTo (P.name)

"C" compareTo ("B")

67 > 66 | 67 - 66

+1 | +1

RHS

③ t.add (P3);

P3.compareTo(P1)

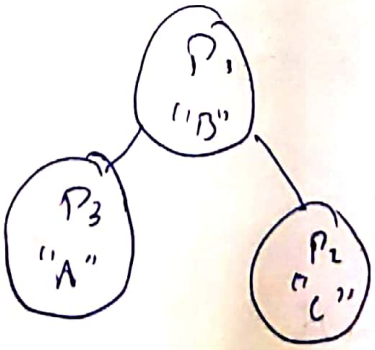
this.name.compareTo(P.name)

"A".compareTo("B")

65 < 66 | 65 - 66

-1 | -1

LHS



Comparator

-> Comparator is a Pre-defined Interface present in java.util Package
-> Comparator was introduced from JDK 1.2
-> Comparator is used for Comparing the objects & Sorting them either in Ascending order (or) Descending order.
-> Comparator is used for Custom Sorting & un-natural Comparision
-> Comparator as an Abstract Method called as Compare ~~method~~ ()

Syntax: Public int Compare (Object o1, Object o2)
↓ Element type e1 ↓ Element e2
↓ Object to be Inserted ↓ Already existing object

Rules for comparing the objects using Comaparator interface :-

1. Design a new class which implements Comaparator interface & implent it java.util Package.
2. Specify Generics by indicating the objects to be Compare
3. override Compare() by specifying the business Logic of Comparition
4. Creat an Object of the class which as the Sorting Logic & pass the reference variable to the Constructor of TreeSet

Example:-

```
Package customSorting;
class Student {
    int age;
    String name;

Student (int age, String name){
    this.age = age;
    this.name = name;
}
@Override
Public String toString (){
    return "Age:" + this.age + "Name:" + this.name;
}
```

```
import java.util.Comparator;

class SortStudentByAge implements Comparator<Student> {
    @Override
    Public int Compare(Student x, Student y) {
        return x.age - y.age;
    }
}

import java.util.Comparator;

class SortStudentByName implements Comparator<Students> {
    @Override
    public int Compare(Student x, Student y) {
        return x.name.compareTo(y.name);
    }
}

import java.util.TreeSet;

class SortStudent {
    Public Static Void main(String[] args) {
        Student S1 = new Student(20, "C");
        Student S2 = new Student(21, "A");
        Student S3 = new Student(19, "B");
    //TreeSet<Student> t = new TreeSet<Student>(new SortStudentByName());
    SortStudentByAge age = new SortStudentByAge();
    TreeSet<Student> t = new TreeSet<Student>(age);
        t.add(S1);
        t.add(S2);
        t.add(S3);
        for (Student Std : t) {
            Sop(Std);
        }
    }
}
```

o/p

Age : 19 Name : B
Age : 20 Name : C
Age : 21 Name : A

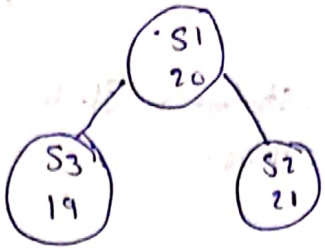
Tracing:

(1) t.add(s1);

(2) t.add(s2);

compare(x, y);
x y
s2 s1

x.age - y.age => 21 - 20 => +1



(3) t.add(s3);

compare(x, y);
x y
s3 s1

x.age - y.age
19 - 20
-1 (Less)

Write a java Program to Compare Mobile object based on Cost & Sort in descending Order. Using Comparator Interface

```
class Mobile {
    int cost;
Mobile (int cost){
    this.cost = cost;
}
  @Override
Public String toString(){
  return "The cost of Mobile is:" this.cost
}}

class MainClass {
  P S v m ( ) {
  Mobile m1 = new Mobile (25,000);
  Mobile m2 = new Mobile (35,000);
  Mobile m3 = new Mobile (15,000);
SOrt MobileCat cost = new Sort MobileCost();
```

```
TreeSet<Mobile> t = new TreeSet<Mobile>(new Cost());
    t.add(m1);
    t.add(m2);
    t.add(m3);

for(Mobile m : t)
  sop(m)
}
import java.util.Comparator;
class SortMobile Cost implements Comparator<Mobile> {
   @Override
    Public int Compare(Mobile x, Mobile y){
       return x.cost - x.cost;
    }
}
```

```
class Employee {
   int id;
   String name;
   Double Salary;

Employee(int id, String name, double Salary){
    this.id = id;
    this.name = name;
    this.Salary = Salary;
}

@Override
Public String toString() {
   return "Employee [id:"+id+", name:"+name+", Salary:"+Salary+"]";
}
}
```

o/p

Employee [id:300, name:A, Salary=34.1]
Employee [id:200, name:C, Salary:54.5]
Employee [id:100, name:B, Salary:76.6]

```
import java.util.Comparator;

class SortEmployeeById implements Comparator<Employee> {
    @Override
    public int compare(Employee x, Employee y){
        return x.id - y.id;
    }
}
```

```
import java.util.Comparator;

class SortEmployeeByName implements Comparator<Employee>{
    @Override
    public int compare(Employee x, Employee y){
        return x.name.compareTo(y.name);
    }
}
```

```
import java.util.Comparator;

class SortEmployeeBySalary implements Comparator<Employee>{
    @Override
    public int compare(Employee x, Employee y){
        return x.salary.compareTo(y.salary);
    }
}
```

```
import java.util.TreeSet;

class SortEmployee {
    public static void main(String[] args){
// TreeSet<Employee> t = new TreeSet<Employee>(new SortEmployeeById());
// TreeSet<Employee> t = new TreeSet<Employee>(new SortEmployeeByName());
  TreeSet<Employee> t = new TreeSet<Employee>(new SortEmployeeBySalary());
        t.add(new Employee(200, "C", 34.5));
        t.add(new Employee(300, "A", 34.1));
        t.add(new Employee(100, "B", 34.6));
        for (Employee emp : t){
            System.out.println(emp);
        }
    }
}
```

Collection

-> Singleton Design Pattern / Singlton class:

① The Process of Creating a Single object or a Single instance of a class is called as Singletonclass.

Rules for developing Singleton class:

① Declare a Private Constructor ( )

② Declare a Public Static helper method which will help to create a Single instance. (& return back the object)

③ Declare a Private Static non-Parameterized Reference Variable.

```
-> Class Account {
    Private Static Account obj;
    Private Account (){
        sop ("object created");
    }
    Public Static void Create object (){
        if (obj == null){
            obj = new Account ();
        }
        else {
            sop ("object cannot be created");
        }
    }
}
```

op

object created
object cannot be created
object cannot be created

```
Class Person {
    ps vm ( ) {
        Account. Create object ();
        Account - Create object ();
        Account . Create object ();
    }
}
```

→ class PrimeMinister {

String name = "Mr. Narendra Modi";

Private Static PrimeMinister pm = null;

Private PrimeMinister() {

Sop ("PrimeMinister Elected");

}

Public Static PrimeMinister getInstance() {

if (pm == null) {

pm = new PrimeMinister();

}

return pm; // return new PrimeMinister();

}

}

o/p PrimeMinister Elected
Name: Mr. Narendra Modi

class Citizen {

P S v m (String[] args) {

// PrimeMinister pm = new PrimeMinister();

PrimeMinister PM = PrimeMinister.getInstance();

Sop ("Name:" + pm.name);

}

}

→ class AadhaarCard {

String name = "Ron"

int aadharNo = 1234; Private Static AadhaarCard obj;

~~Private Static AadhaarCard AD = null; Private Static AadhaarCard AD = 0;~~

Private AadhaarCard() {

Sop ("AadhaarCard Verified");

}

Public Static AadhaarCard getInstance() {

if (AD == null) {

AD = new AadhaarCard();

}

return AD;

}

}

```
class user {
  p svm( ) {
  // Aadhaar ac = new Aadhar Card();
  AadhaarCard ac = AadhaarCard.getInstance();
  Sop(ac.aadhaarNo+" "+" "+ ac.name);
  }
}
```

->
```
class Marriage {
  String bride = Aditi";
  String groom:   "SAGAR";

  Private Static Marriage obj;

  Private Marriage() {
     Sop ("Getting Married");
  }

  Public Static Marriage getInstance() {
       if (obj == null){
          obj = New Marriage();
       }
       return obj;
    }
}

Class Hall {
    Psvm ( ) {
  Marriage m = Marriage.getInstance();
  Sop(m.Groom+" weds"+ m.bride);
  }
}
```

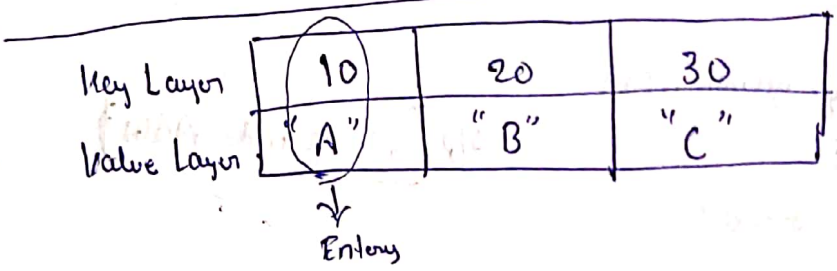
o/p SAGAR weds Aditi

MAP :-

-> MAP is a Part of Collection framework which doesn't Extend Collection Interface
-> MAP is also a Pre-defined Interface Present in java.util Package
-> MAP was introduced from JDK 1.2
-> The Objects inside Map is organized in the form of Key & Value Pair where in Key Cannot be duplicated & Values can be duplicated.
-> 1 Key & Value together is Referred as Entry, there Fore MAP is a Collection of entries.

The implementation classes of the MAP interface are as Follows:

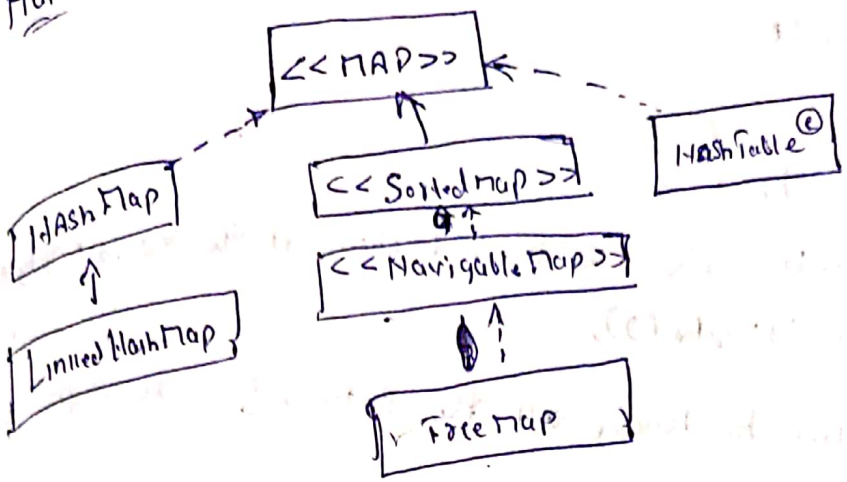
1. HASH MAP
2. LINKED HASH MAP
3. TREE MAP
4. HASH TABLE



Important Methods Present in MAP interface:

1. Put()
2. get()
3. ContainsKey()
4. Contains value()
5. Size()
6. isEmpty()
7. Clear()
8. Remove()
9. Public Set<E> keySet();

Map Hierarchy:




```
import java.util.HashMap;

class Demo{
  psvm (String[] args){
    HashMap hm= new HashMap();

  // put() is used to add one entry into the map
    hm.put(10, "Java");
    hm.put("Sql", 12.4);
    hm.put(1.2, 500);

    sop (hm);

  sop ("----");

// size() is used to return the no of entries present
    sop (hm.size());

    sop ("----");

// get() is used to return the value based on the key
    sop (hm.get(12));

  sop("----");

// containsKey() is used to find if the key is present or not
      sop (hm.containsKey("Sql"));

      sop ("----");

// containsValue() is used to check if the value is present or not
      sop (hm.containsValue("Java"));

      sop ("----");
```

o/p:

{1.2=500, 10=Java, Sql=12.4}

3
---
null
---
true
---
true

{1.2=500, 10=Java, Sql=12.4}
{1.2=500, Sql=12.4}
---
false
true

```
Sop (hm);
// remove() is used to remove the entry based on key
    hm.remove(10);
    Sop (hm);
    Sop ("....");
// isEmpty() is used to check if the map is Empty or not
    Sop (hm.isEmpty());
// clear() is used to remove all the entries from the map
    hm.clear()
    Sop (hm.isEmpty());
} }
```

-> import java.util.LinkedHashMap;
import java.util.Set;
import java.util.TreeMap;

```
class Test {
    p s v m (String [] args){
LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer, String>();
    lhm.put (10, "Apple");
    lhm.put (20, "Banana");
    lhm.put (30, "Mango");

Set<Integer> s = lhm.keySet(); // returns a set view of keys
  for (int key : s){
      Sop (key + " -> " + lhm.get(key));
  }
    Sop ("- - -");
TreeMap<Double, String> t = new TreeMap<Double, String>();
    t.put (2.5, "A");
    t.put (3.2, "B");
    t.put (1.7, "C");
```

```
Set<Double> setOfKeys = t.keySet();
for(double key : setOfKeys){
    Sop(key + " " + t.get(key));
}
}
```

10 -> Apple
20 -> Banana
30 -> Mango

1.1	C
2.5	A
3.2	B

```
import java.util.LinkedHashMap;
class Demo {
    psvm( ) {
LinkedHashMap l = new LinkedHashMap();
    l.put(10, "Sony");
    l.put(20, "Apple");
    Sop(l);
    l.put(20, "Samsung");
    Sop(l);
}}
```

o/p
{10=Sony, 20=Apple}
{10=Sony, 20=Samsung}

	HashMap©	LinkedHashMap©	TreeMap©
Package	java.util	java.util	java.util
Insertion order	X	✓	X
Sorted Order	X	X	✓
Null as a key	Yes	Yes	No
JDK Version	JDK 1.2	JDK 1.4	JDK 1.2

Hashtable
- DNC
- JDK 1.0
- java.util
- Thread Safe / Synchronized

-> Legacy Collection

Hashmap
- DNC
- JDK 1.2
- java.u1.1
- Not Thread Safe / Not Synchronised

Comparable

-> Class Student implements Comparable<Student> {

```
int age;

Student (int age) {
    this.age = age;
}

@Override
Public String toString() {
    return "Age: " + this.age;
}

@Override
    Public int compareTo(Student s) {
        return this.age - s.age;
    }
}
```

```
import java.util.TreeSet;
import java.
```

```
Class SortingUsingTreeSet {
    Psvm() {
        Student s1 = new Student(81);
        Student s2 = new Student(84);
        Student s3 = new Student(82);

        TreeSet<Student> t = new TreeSet<Student>() {
            t.add(s1);
            t.add(s2);
            t.add(s3);
            for (Student std : t) {
                Sop(std);
            } } }
```

```
import java.util.ArrayList;
import java.util.Collections;

class SortingUsingArrayList {
    psvm() {
        Student s1 = new Student(45);
        Student s2 = new Student(48);
        Student s3 = new Student(46);

        ArrayList<Student> l = new ArrayList<Student>();
        l.add(s1);
        l.add(s2);
        l.add(s3);

        Sop("Before Sorting:");
        for(Student std : l){
            Sop(std);
        }

        Collection.sort(l);

        Sop("After Sorting:");
        for(Student std : l){
            Sop(std);
        }
    }
}
```

o/p:

Before Sorting:
Age: 81
Age: 84
Age: 82
After Sorting:
Age: 45
Age: 46
Age: 48

Explanation

```
-> class Employee {
    int id;
    Employee(int id) {
        this.id = id;
    }
    @Override
    Public String toString()
    return "Id" + this.id;
}
```

```
class SortEmployeeById implements Comparator<Employee>{
    @Override
    public int Compare(Employee x, Employee y){
        return x.id - y.id;
    }
}
```

(Sorting using TreeSet)

```
import java.util.TreeSet;
class SortUsingTS{
    psvm(String[] args){
    TreeSet<Employee> t = new TreeSet<Employee>(new SortEmployeeById());
        t.add(new Employee(200));
        t.add(new Employee(300));
        t.add(new Employee(100));
        for(Employee emp : t){
            sop(emp);
        }
    }
}
```

(Sorting using ArrayList)

```
import java.util.ArrayList;
import java.util.Collections;
class SortUsingAL{
    psvm(String[] args){
    ArrayList<Employee> l = new ArrayList<Employee>();
        l.add(new Employee(200));
        l.add(new Employee(300));
        l.add(new Employee(100));
        Collections.sort(l, new SortEmployeeById());
        for(Employee emp : l){
            Sop(emp);
        }
    }
}
```

o/p
id : 100
id : 200
id : 300

Comparing Objects & Sorting in Asc / Des order

<< Comparable >>
- DOI in java.lang (JDK 1.2)
- compareTo()
- Default Sorting
- Mutual Comparison
- Source Code & Sorting Logic in Same class (Dependent)

<< Comparator >>
- DOI in java.util (JDK 1.2)
- Compare()
- Custom Sorting
- Unmutual Comparision
- Source Code & Sorting logic are in different class (Independent)

Note: addAll() is use to Add all the objects of one Collection into another Collection

-> removeAll() is use to remove all the objects of one Collection from another Collection

-> containsAll() is used check if one Collection is having all the objects of another Collection

-> import java.util.*;

```
class Solution {
    psvm ( ){
    ArrayList l = new ArrayList ();
      l.add (10);
      l.add (20);
    LinkedList z = new LinkedList();
      z.addAll (l);
      z.add (30);
    Sop (z.containsAll (l));
      Sop (z);
      z.removeAll (l);
}}  Sop (z);
```

o/p
[10, 20, 30]
[30]

Page No.
Date

18/12/21

Exception Handling.

Errors:- Error is a Mistake or a Problem which occurs During the Execution of a Program.

-> We get Compilation Error (or) Compile time Error due to Syntax (or) Syntactical Mistake.

-> We get Run-time Error when we Execute a class without main method & also when the Stack is full.

-> Errors Should always be debug.

Exception :- Exception is an Event or interruption which terminates the Execution of a Program. Where in the below lines of code will not be Executed.

-> Exception can also be refered as a Run-time interuption ~~And~~ also which Stops the Program Execution.

-> Therefore Exceptions Should be handled & Errors Should be debug.

-> The Process of handling an Exception is called as Exception handling.

try and Catch block:-

-> Exceptions are Generally handled with the help of try & catch block.

-> The Critical lines (or) Problematic lines which gives an Exception. Should be written inside try block.

-> The Solution for the Exception occurred should be written inside catch block.

-> try block & catch block should always be together.

-> Catch block gets executed only when an Exception occurs

Syntax:

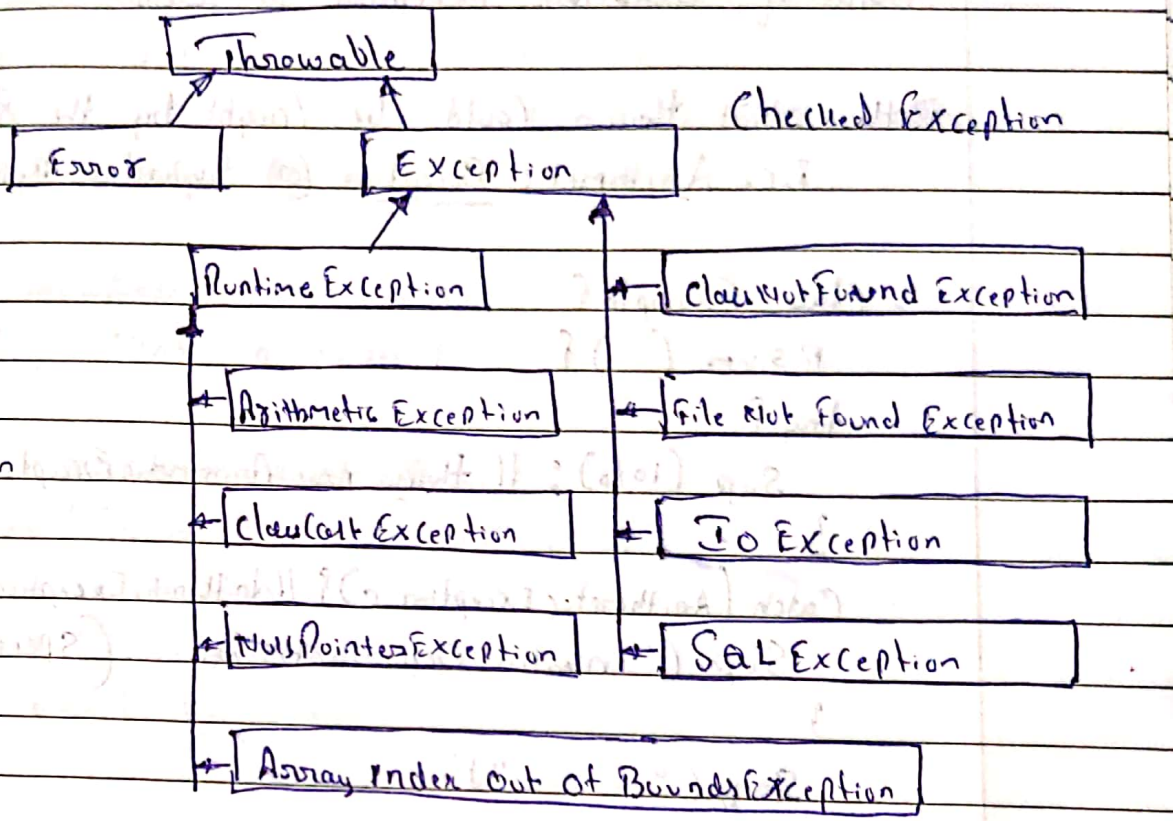
```
try {
                                    (copy the Exception Line)
}
Catch (ExceptionName ReferenceVariable) {
                                    {
}
```

->
```
import java.util.Scanner;
class Demo {
    public static void main (String[] args ){
        Sop("Start");
      Scanner Scan = new Scanner (System.in);
       Sop ("Enter first no:");
        int x = Scan.nextInt();
        Scan.close();
       try {
         Sop (x/4);
       }
       Catch (ArithmeticException e) {
         Sop ("Invalid Denominator");
       }
        Sop ("End");
}
}
```

```
O/P Start
Enter first no:
10
Enter Second no:
0
Invalid Denominator
End
```

Page No.
Date

Exception Hierarchy



Note: All the Exceptions are either Pre-defined (or) User defined classes

→ It is always a Good Practice to handle an Exception using Suitable try and Catch block

```
class Test {
  PSVM () {
    Sop("Start");
    int[] a = {12, 34, 56};
    try {
      Sop(a[100]);
    }
    catch (ArrayIndexOutOfBoundsException x) {
      Sop ("Invalid Index");
    }
    Sop ("End");
  }
}
```

o/p

```
Start
Invalid Index
End
```

Page No.
Date

Note:- When we divide a number by Zero(0) internally an object of arithmetic Exception is thrown

→ the object thrown could be caught by the Same type i.e. Arithmetic Exception (or) Superclass type i.e. Exception

```
-> class Example {
    psvm ( ) {
    try {
      Sop (10/0); // throw new ArithmeticException ();
    }
    Catch (ArithmeticException e) { // ArithmeticException e = new ArithmeticException();
      Sop ("Invalid Denominator"); (Specialization)
    }
      Sop ("------");

    try {
      Sop (10/0); // throw New ArithmeticException ();
    }
    Catch (Exception e) { // Exception e = new ArithmeticException ();
      Sop ("Invalid Denominator"); (Generalization)
    }
}
```

Note:

① One try block can have any number of Catch blocks & that Suitable ~~Suitable~~ Catch block will get Executed.

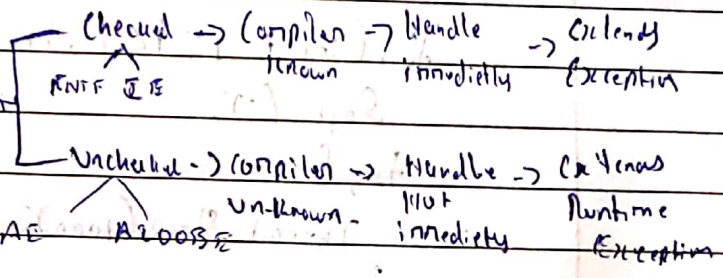
② It is always a Good Practice to handle the Super class catch block as the last Catch block.

```
->Class Solution {
    Psvm ( ) {
    try {
      Sop (10/0);
    }
    Catch (ArrayIndexOutOfBoundsException x) {
      Sop ("Invalid Index");
    }
    Catch (ArithmeticException e) {
      Sop ("invalid Denomination");
    }
     Catch (Exception e){
       Sop ("handled");
     }
    }
```

O/P
Invalid Denomination

Types of Exception :-

1. Check Exception
2. Un-checked Exception



1. Check Exception :- check exceptions are Compiler known Exceptions and the Compiler will force you to handle immediately

-> All check Exception will inherit Exception Class.

2. Unchecked Exception :- unchecked Exception are Unknown to the Compiler And the Compiler will not force you to handle immediately

Page No.
Date

-> All unchecked Exceptions will inherit Runtime Exception class

Examples for unchecked Exception :-

```
-> class Demo{
    Psvm (){
    Sop (1010);
    int[] a = {10,20,30};
    Sop (a[99]);
    }
}
```

Examples for checked Exception:

```
-> class A{
    Psvm (){
    for(int i=1; i<=5; i++){
    Sop (i);
    try{
    Thread.Sleep (1000);
    }
    Catch (InterruptedException e){
    Sop ("Some Problem");
    }
}}
```

Page No.
Date

-> import java.io.FileNotFoundException;
import java.io.FileReader;
class B {
psvm () {
try {
FileReader f = new FileReader ("~~Goto~~ Sagar.txt");
}
catch (FileNotFoundException e) {
Sop ("File not Present");
}}}

O/P
File not found

throws :- throws is a key word which is used to indicate the Caller about the possibility of an Exception

-> throws can be used to a methods & Constructors but majorly used with methods.

-> throws can be used with both check Exceptions & unchecked Exceptions but majorly used with check Exceptions.

-> throws is used with Method declaration & it is possible to use multiple Exceptions with the help of throws.

-> class Test {
Static void div() throws ArithmeticException {
Sop (1010);
}
psvm ( ) {
Sop ("Start");

try {
div (); // Caller
}
catch (ArithmeticException e) {
Sop ("Invalid Denomination")
}
Sop ("End")
}}

O/P
Start
Invalid Denomination
End

② -> class Example {
static void display() throws Exception {
for (int i=1; i<=5; i++) {
Sop(i);
Thread.sleep(1000);
}
}

Psvm () {
try {
display();
}
catch (Exception e) {
Sop ("Some Problem");
}
}}

O/P
1
2
3
4
5

③ -> import java.io.*;
class Solution {
void read() throws FileNotFoundException {
FileReader f = new FileReader("dinga.txt");
}
Public static void Main (String[] args) {
Solution s = new Solution();
try {
s.read();
}
Catch (FileNotFoundException e) {
Sop ("File not Present");
}
}}

O/P File not Present

Page No.
Date

Note: we can declare multiple exception with throws

void m1() throws ArithmeticException, ClassCastException {
}

(or)

void m1() throws Exception {
}

Note: throws will avoid the mis-use of costly resources
-> we should handle an exception only when it is necessary, un-necessary handling of exception should be avoided.
-> in other throws is use to transfer the exception to the caller side
-> we can use throws with main(), but it is not a good practice because the caller of the main() is JVM & we cannot handle the exception at the JVM side.

finally block:-

-> finally block is a set of instructions or a block of code which gets executed always irrespective of an exception occurs or not

Syntax:-

```
finally
{
  ≡
}
```

Page No.
Date

```
class FinallyBlockDemo{
  Psum (){
    Sop ("Start");
  try {
    Sop (10/0);
  }
  Catch (ArithmeticException e){
    Sop ("Invalid Denominator");
  }
  finally{
    Sop("Inside Finally Block");
    }
    Sop ("end");
  }
}
```

O/P
Start
Invalid Denominator
Inside Finally Block

Note: We can have nested try & catch block

(2) It is possible to have try & catch block within finally block

(3) We cannot have any executable lines of code b/w try & catch block

```
try {
  try{
  }
  // int a=10;
  catch (Exception e){
  }
}
Catch (ArithmeticException e){
}
```

```
finally{
  try{
  }
  Catch (Exception e2){
  }
}
```

Page No.
Date

Note: It is possible to have try block with at Catch block when have try & finally block

Syntax:

```
try{
}
finally{
}
```

Important Methods with reflect to Exception:-

-> getMessage() :- This method is used to return a small msg about the Exception

Syntax:

```
public String getMessage()
```

-> class MainClass {

```
psum ( ) {
try {
  Sop (10/0); // throw new ArithmeticException ();
}
catch (Exception e) { // Exception e = new ArithmeticException();
  Sop (e.getMessage());
  String message = e.getMessage();
  Sop (message);
  }
}
}
```

o/p
/ by zero
/ by zero

Page No.
Date

⇒ printStackTrace(): This method is used to print the complete information about the exception

That is:
① Exception name
② Line number
③ Small message about the exception

```
→ class MainClass {
    P S V M (){
    Sop ("START");
    try {
      Sop (10/0);
    }
    Catch (Exception e) {
      e.printStackTrace ();
    }
    Sop ("END");
  }
}
```

o/p

```
START
java.lang.ArithmeticException: / by zero
    at org.mainclass.main(mainclass.java:)
END
```

Custom Exception (or) User defined Exception:-

Based on certain projects it is sometimes necessary to create an Exception related to the project, therefore any exception which is created by the user or programmer is called as Custom Exception (or) User defined Exception.

Page No.
Date

Rules for developing Custom Exception (or) User defined Except:-

1. Create a class based on the Exception name.
2. To create a Check Exception class should inherit Exception class and to create unchecked Exception class should inherit Run-time-Exception class.
3. optionaly we can override getmessage()
4. invoke the Exception object using throw keyword, handle the Exception using Suitable try & catch blocks & provide Suitable Solution for the Exception occured

throw:-

1. throw is a keyword which is used to invoke an object of Exception type
2. throw is generally used with Custom Exception

Syntax: throw new ExceptionName();
or
throw objectOfExceptionType;

Unchecked

=> class InvalidPasswordException extends RuntimeException {

}

```
import java.util.Scanner;
class Login {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        sop("Enter Password:");
        int password = scan.nextInt();
        scan.close();
```

```
if (password == 123){
  Sop ("Login Successfull");
}
else {
  try {
    throw new InvalidPasswordException();
  }
  Catch (InvalidPasswordException e) {
    Sop ("Please Enter Valid Password");
  }
}
}
```

o/p

Enter Password:
126
Please Enter Valid Password

② Checked Exception

```
-> Class InsufficientBalanceException extends Exception {
    Private String message;
    InsufficientBalanceException (String message) {
      this.message = message;
    }
    @Override
    Public String getMessage () {
      return this.message;
    }
}
```

o/p

```
Welcome To Atm
-----
Enter Amount to be withdrawn:
4500
Insufficient Funds !
```

```
import java.util.Scanner;
Class Atm {
  PSVM () {
    Sop ("Welcome To Atm");
    Sop ("-----");
    int balance = 5000;
```

```
Scanner Scan = new Scanner(System.in);
Sop ("Enter Amount to be withdrawn:");
int amount = Scan.nextInt();
Scan.close();

if(amount <= balance){
    Sop ("Amount withdrawn Successfully");
}
else{
  try{
throw new InsufficientBalanceException("Insufficient Funds!");
    }
    Catch (Exception e) {
        Sop (e.getMessage());
    }
    }
} }
```

(3) Write a java Program to Create a Check Exception Called AgeInvalidException and Override GetMessage()

```
Class AgeInvalidException Extends Exception {
    Private String Message;
AgeInvalidException (String Message){
    this.Message = Message;
}
@Override
Public String GetMessage(){
    Return this.Message;
}
}
```

```
import java.util.Scanner;
class MatrimonyPortal {
  Psvm( ) {
  Sop ("welcome to Shaadi.com");
  Scanner Scan : new Scanner (System.in);
  System.out.println ("Enter Age:");
  int age = Scan.nextInt();
  Scan.close ();

  if (age >=21){
    Sop ("Get Married, get lost");
  }
  else {
    try {
// Age InvalidException obj: new AgeInvalidException ("Enjoy Job, later Marriage");
    throw obj; //.
    throw new AgeInvalidException ("Have Patience");
  }
  catch (AgeInvalidException e) {
    Sop (e.getMessage());
}}}
```

O/P

Welcome to Shaadi.com
Enter Age:
20
Have Patience