Durgasoft SCJP Notes

Part-2

12/03/11

Collection framework

→ An Array is an indexed Collection of fixed no. of homogeneous data elements.

* Limitations of object arrays :-

(1) Arrays are fixed in size. i.e, once we created an array there is no chance of increasing or decreasing size based on our requirement. Hence, to use arrays concept compulsory we should know the size in advance, which may not possible always.

(2) Arrays can hold only Homogeneous data elements. i.e, (Same type)

ex!- Student[] s = new Student[1000];

s[0] = new Student[]; ✓

s[1] = new Student[]; ✓

s[2] = new Customer[]; ✗ CE!- Incompatible types
found : Customer
required : Student.

↳ But we can resolve this problem by using Object type arrays.

ex!- Object[] a = new Object[1000];

a[0] = new Student[]; ✓

a[1] = new Customer[]; ✓

(3) Arrays concept not built based on some datastructure. Hence readymed method support is not available for every requirement. Compulasary programmer is responsible to write the logic.

→ To resolve the above problems SUN people introduced Collections Concept.

→ Advantages of Collections over arrays :-

(1) Collections are growable in nature. Hence based on our requirement we can increase or decrease the size.

(2) Collections can hold both homogeneous & heterogeneous objects.

(3) Every Collection class is implemented based on some datastructure. Hence readymed method support is available for every requirement.

dis. of Collections :-

→ Performance point of view Collections are not recommended to use. This is the limitation of Collections.

difference b/w arrays & Collections :-

Array	Collections(AL, VL, LL - - - -)
1) Arrays are fixed in size	1) Collections are growable in nature
2) Memory point of view arrays Concept is not recommended to use	2) memory point of view Collections Concept is highly recommended to use.
3) Performance point of view arrays Concept is highly recommended to use.	3) Performance point of view Collections is not recommended to use.
4) Arrays can hold only homogeneous data elements.	4) Collections can hold both homogeneous & Heterogeneous objects.
5) There is no underlying d.s for arrays. Hence readymed method support is not available	5) Underlying D.S is available for every Collection class. Hence readymed method support is available.

→ Arrays can be used to hold both primitives & objects.	→ Collections can be used to hold only objects but not for primitives.

Collection :-

→ A group of individual objects as a single entity is called Collection

Collection framework :-

→ It defines several classes & Interfaces, which can be used to represent a group of objects as a single Entity.

Terminology :-

Java	C++
Collection	→ Containers
Collection framework	→ STL (Standard Template Library)

9-Key interfaces of Collection frame work :-

① Collection (Interface) :

→ If we want to represent a group of individual objects as a single Entity then we should go for Collection.

→ In general Collection Interface is considered as root Interface of Collection framework.

→ Collection Interface defines the most common methods which can be applicable for any Collection object.

Collection vs Collections :-

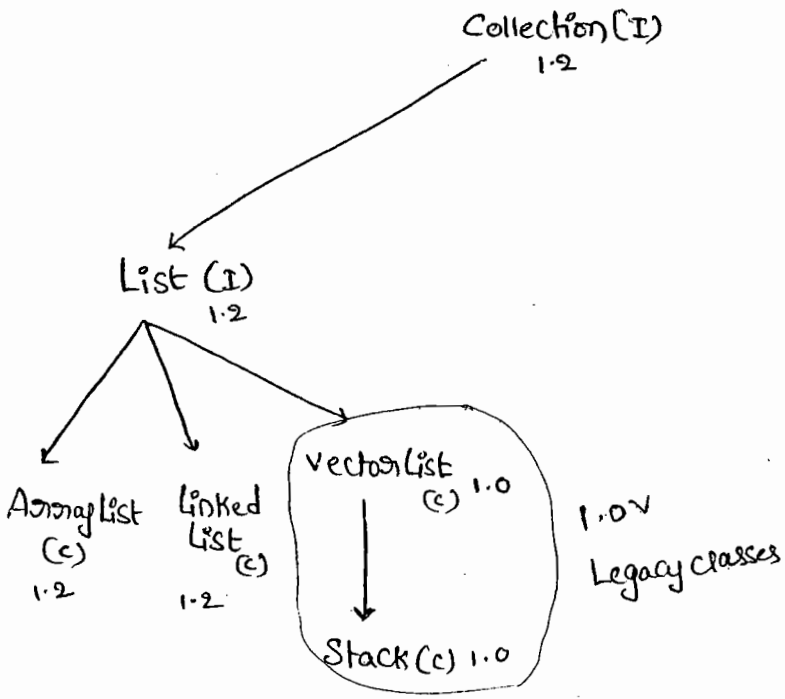
→ Collection is an interface, can be used to represent a group of individual object as a single Entity. where as,

→ Collections is an Utility class, present in java.util package, to define several utility methods for Collections.

3) List (Interface) :-

→ It is the child Interface of Collection.

→ If we want to represent a group of individual objects where insertion order is preserved & duplicates are allowed. Then we should go for List.

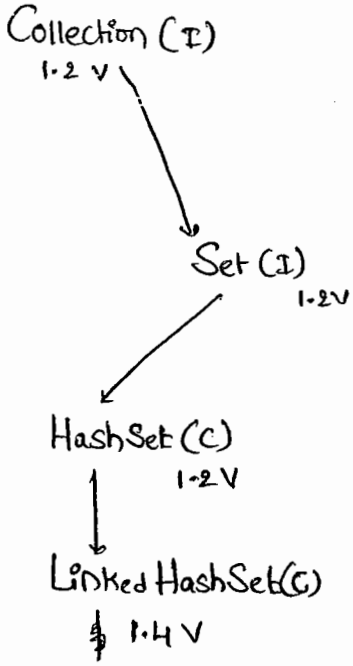


→ Vector & Stack classes are re Engineered in 1.2 version to fit into Collection framework.

③ Set (Interface) :-

→ It is the child Interface of Collection.

→ If we want to represent a group of individual objects where duplicates are not allowed & insertion order is not preserved. Then we should go for "Set".



④ SortedSet (I) :-

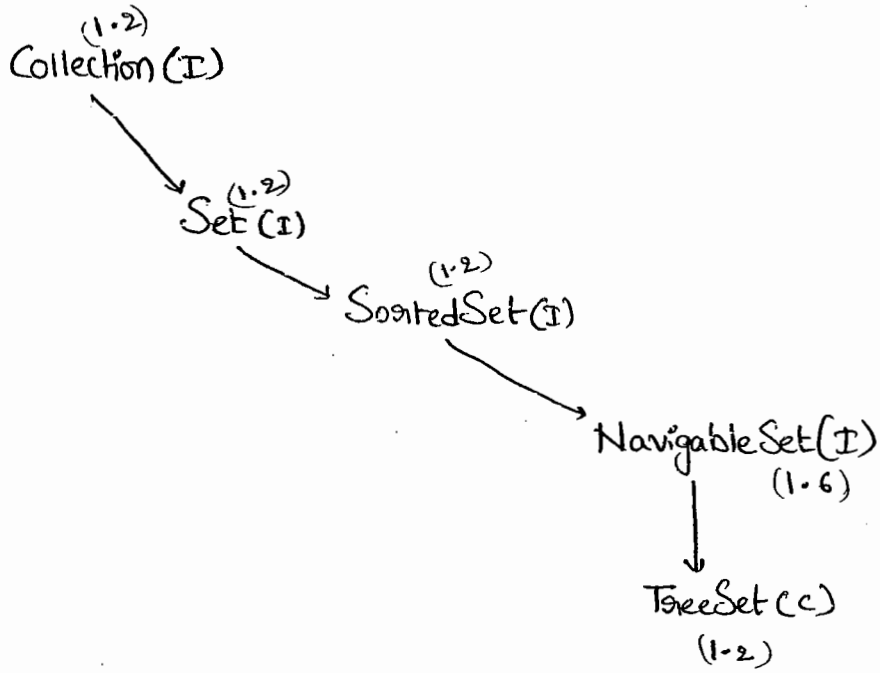
→ It is the child interface of Set.

→ if we want to represent a group of ~~unique~~ individual objects, according to some sorting order. then we should go for SortedSet.

⑤ NavigableSet (I) :.

→ It is the child interface of SortedSet, to provide several methods for Navigation purposes.

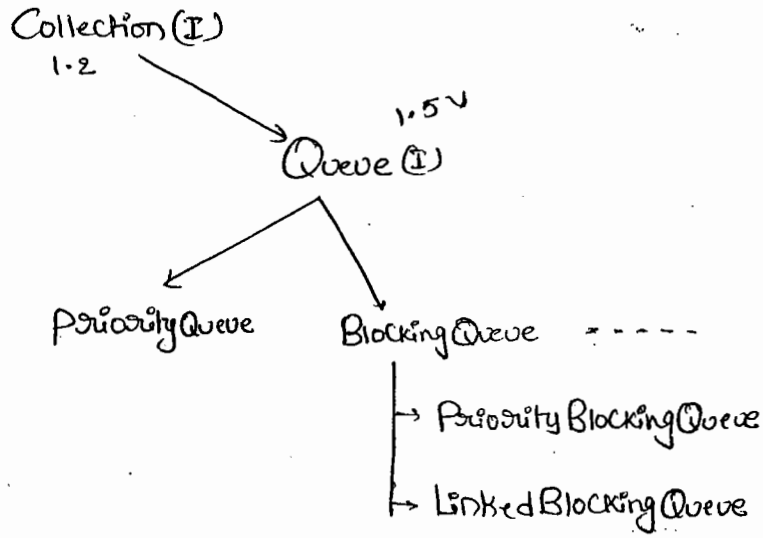
→ It is introduced in 1.6 version.



⑥ Queue (I) :- (1.5V)

→ It is the child Interface of Collection.

→ If we want to represent a group of individual objects, prior to processing then we should go for Queue.



Note :-

→ all the above Interfaces (Collection, List, Set, SortedSet, NavigableSet, Queue) ment for representing a group of individual objects.

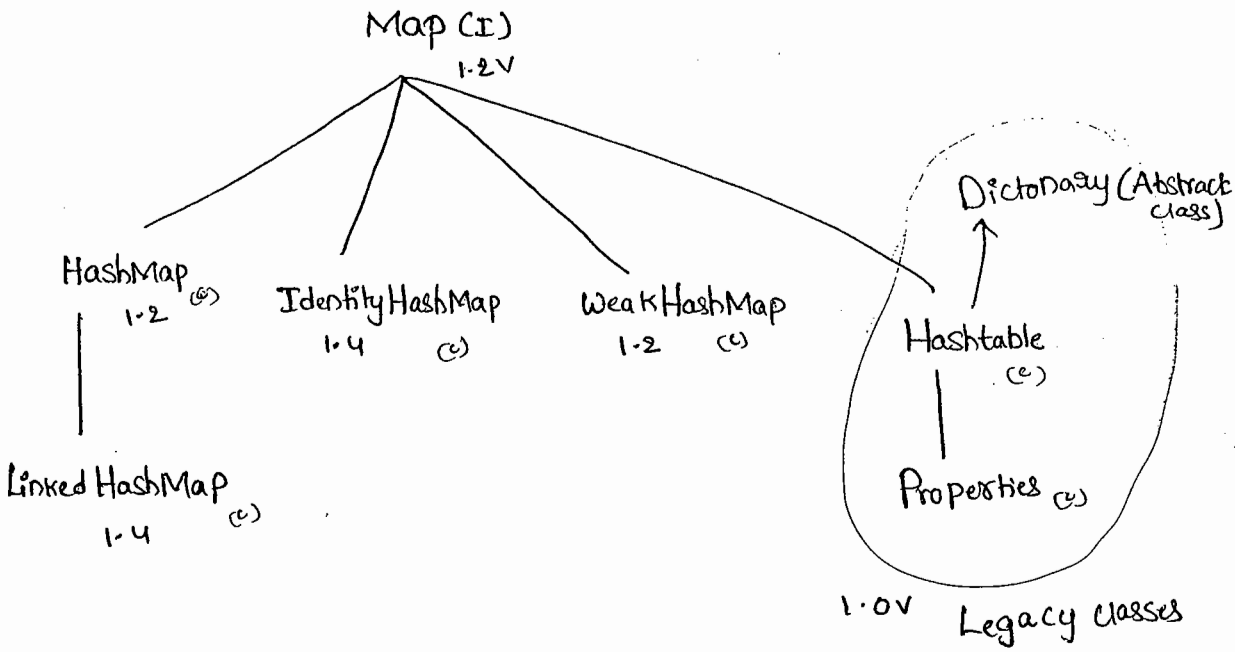
→ If we want to represent a group of objects as key-value pairs Then we should go for 'Map'.

(7) Map(I):-

→ If we want to represent a group of objects as Key-value pairs then we should go for Map.

→ Both Key & value are objects only.

→ duplicate Keys are not allowed, But values Can be duplicated.



Note:-

⟹ Map is not child Interface of Collection.

(8) SortedMap (I):-

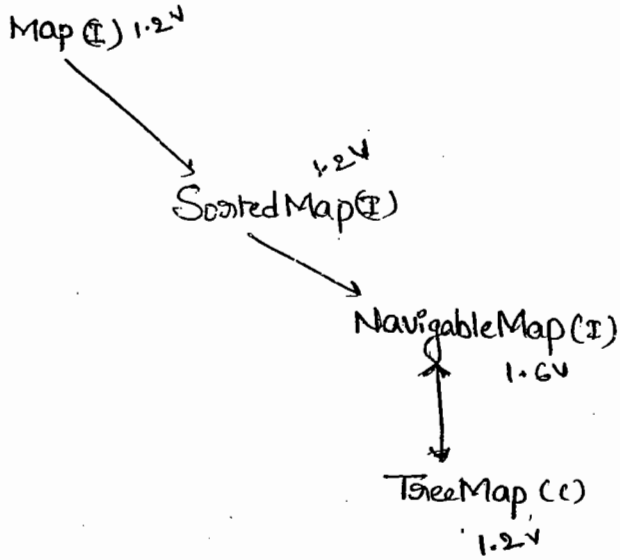
→ If we want to represent a group of objects as Key-value pairs according to Some Sorting Order. Then we Should go for SortedMap.

→ Sorting should be done only based on Keys. but not based-on values.

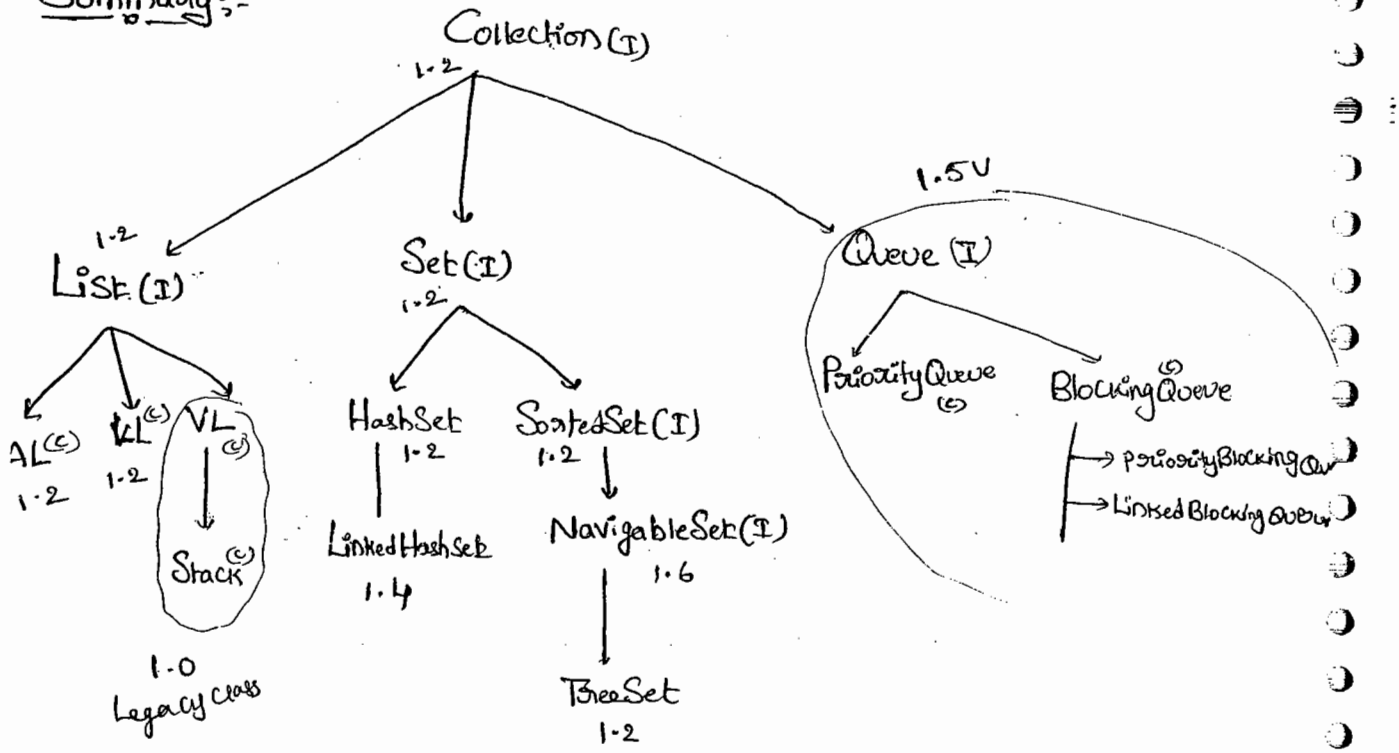
→ SortedMap is child Interface of Map.

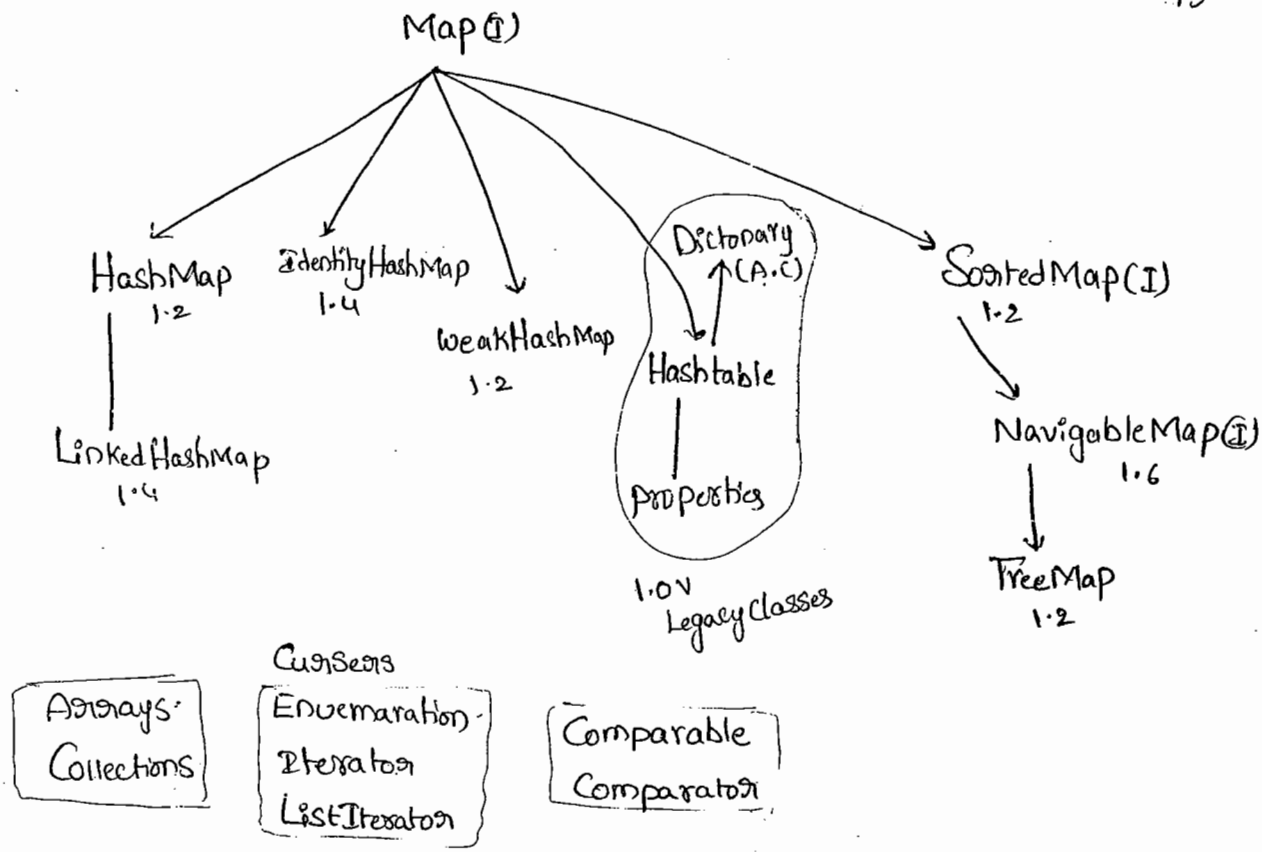
(9) NavigableMap (I):-

→ It is the child Interface of SortedMap & define Several methods for Navigation purposes.



Summary:-





→ In the Collection-frame work the following are Legacy characters

(1) Enumaration (I)
(2) Dictionary (Ab c)
(3) Vector
(4) Stack
(5) Hashtable
(6) properties

(3) (4) (5) (6) → classes
(1) (2) (3) (4) (5) (6) → 1.0V

Collection framework:

Collection (I):-

→ If we want to represent a group of individual objects as a single entity then we should go for Collection.

→ Collection Interface defines the most common methods which can be applied for any Collection object.

→ The following is the list of methods present in Collection Interface.

① boolean add(Object o)

② boolean addAll(Collection c)

③ boolean remove(Object o)

④ boolean removeAll(Collection c)

⑤ boolean retainAll(Collection c)

→ To remove all objects except those present in c.

⑥ void clear()

⑦ boolean isEmpty()

⑧ int size()

⑨ boolean contains(Object o)

⑩ boolean containsAll(Collection c)

⑪ Object[] toArray()

⑫ Iterator iterator()

(2) List (I) :-

→ List is the child Interface of Collection.

→ If we want to represent a group of individual objects where duplicate objects are allowed & insertion order is preserved. Then we should go for List.

→ Insertion order will be preserved by means of Index.

→ We can differenciate duplicate objects by using Index. Hence Index plays a very important role in List.

→ List Interface defines the following methods

1. boolean add(int index, Object o)
2. boolean addAll(int index, Collection c)
3. Object remove(int index)
4. Object get(int index)
5. Object set(int index, Object new)
   old
6. int indexOf(Object o)
7. int lastIndexOf(Object o)
8. ListIterator listIterator()

It contains 4 classes:-

(i) ArrayList (C) :- 1.2

(ii) LinkedList (C) :- 1.2

(iii) VectorList (C) :- 1.0

(iv) Stack (C) :- 1.0

(1) ArrayList (c) :-

→ The underlying datastructure for ArrayList is Resizable Array (or) Growable Array.

→ Insertion Order is preserved.

→ Duplicate objects are allowed.

→ Heterogeneous objects are allowed.

→ Null insertion is possible.

Constructors :-

(1) `ArrayList AL = new ArrayList();`

→ Creates an Empty ArrayList object, with default initial Capacity 10.

→ Once AL reaches it's max. capacity then a new AL object will be Created with.

$$\text{New Capacity} = \text{Current Capacity} * \frac{3}{2} + 1$$

(2) `ArrayList l = new ArrayList(int initialCapacity);`

→ Creates an Empty Arraylist object with the Specified initial Capacity.

(3) `ArrayList l = new ArrayList(Collection c);`

→ Creates an Equivalent ArrayList object for the Given Collection object

i.e, This Constructor is for dancing b/w Collection objects

Ex:-

```
import java.util.*;
class ArrayListDemo
{
  P.S.V.m(String[] args)
  {
    ArrayList a = new ArrayList();
    a.add("A");
    a.add(10);
    a.add("A");
    a.add(null);
    S.o.pln(a);   [A, 10, A, null]
                   0  1  2   3
    a.remove(2);
    S.o.pln(a);   [A, 10, null]
    a.add(2,"M");  [A, 10, M, null]
    a.add("N");    [A, 10, M, null, N]
    S.o.pln(a);   [A, 10, M, null, N]
  }
}
```

```
S.o.pln(a.size()); //5
a.clear(); // []
a.addAll(a); //[A, 10, M, null, N, A, 10, M, null, N]
```

Note:-

⊛ In Every Collection class toString() is overridden to return its Content directly in the following formatt.

[obj1, obj2, obj3 .....]

↪ Usually we Can Use Collection to Store & transfer Objects. to provide Support for this requirement Every Collection class implements Serializable & Clonable Interfaces.

→ ArrayList & Vector classes implements RandomAccess Interface, so that any random element we can access with same speed. Hence, if our frequent operation is retrivable operation then best suitable data structure is ArrayList. (Advantage)

→ If our frequent operation is Insertion or deletion operation in the middle then ArrayList is the worst choice, because it required several shift operations. (disadvantage).

Q: Differences b/w ArrayList & Vector?

ArrayList	Vector
① No method is Synchronized	① Every method is Synchronized
② multiple threads can access ArrayList simultaneously, hence ArrayList object is not threadsafe	② At any point only one thread is allowed to operate on vector object at a time. Hence vector object is Thread Safe
③ Threads are not required to wait, & hence performance is high.	③ It increases waiting time of threads & hence performance is low.
④ Introduced in 1.2 version & Hence it is non-legacy	⑤ Introduced in 1.0 version & Hence it is Legacy.

Q) How to get Synchronized Version of Arraylist?

A) → By using Collections class SynchronizedList() we can get Synchronized version of Arraylist.

```
Public static List SynchronizedList(List l)
```

Eg:-

```
ArrayList l = new ArrayList();
List l1 = Collections.SynchronizedList(l)
```

l1 → Synchronized

l → NON-Synchronized.

→ Similarly we can get Synchronized Version of Set & Map objects by using the following methods respectively.

① public static Set SynchronizedSet(Set s)

② public static Map SynchronizedMap(Map m)

Note:-

→ If our frequent Operation is Insertion or deletion in the middle then Arraylist is not recommended. to handle this requirement we should go for LinkedList.

i) LinkedList © :-

→ The underlying datastructure is doubleLinkedList.

→ Insertion order is preserved.

→ duplicate objects are allowed.

→ Heterogeneous " "

→ Null insertion is possible.

→ Implements Serializable & Clonable interfaces but not RandomAccess-interfaces.

→ Best Suitable if our frequent operation insertion or deletion in the middle.

→ Worstest choice if our frequent operation is retrival.

Constructors:-

① LinkedList l = new LinkedList();

→ Creates an Empty LinkedList object.

② LinkedList l = new LinkedList(Collection c)

→ for interConversion b/w Collection objects.

LinkedList Specific methods:-

→ Usually we can use Linkedlist to implements Stacks & Queues to support this requirements LinkedList class define the following Six Specific methods.

① Void addFirst(Object o);

② void addLast(Object o);

③ Object removeFirst();

④ Object removeLast();

⑤ Object getFirst();

⑥ Object getLast();

Ex:-

```
import java.util.*;
class LinkedListDemo
{
 p.s.v.m(String[] args)
 {
   LinkedList l = new LinkedList();
   l.add("durga");
   l.add(30);
   l.add(null);
   l.add("durga");
   l.set(0,"Software");
   l.add(0,"venky");
   l.removeLast();
   l.addFirst("ccc");
   S.o.pln(l);
 }
}
```

[durga, 30, null, durga]
[S/w, 30, null, durga]
[venky, S/w, 30, null, durga]
[venky, S/w, 30, null]
[ccc, venky, S/w, 30, null]

ccc venky ~~durga~~ Software 30 null ~~durga~~

[ccc, venkey, Software, 30, null]

ii) Vector(c) :-

→ The underlying datastructure is Resizeable array or growable array.

→ Insertion Order is preserved.

→ duplicate objects are allowed.

→ null insertion is possible

→ Heterogeneous objects are allowed.

→ implements Serializable, Clonable & RandomAccess interfaces.

→ Best Suitable if our frequent operation is Retrival & worstest choice if our frequent operation is insertion or deletion in the middle.

→ Every method in vector is Synchronized. Hence vector object is Threadsafe.

Constructors:-

(1) `Vector v = new Vector();`

→ Creates an Empty Vector object with default initial Capacity 10.

→ Once vector reaches it's Max. Capacity a new vector object will be Created with double Capacity.

New Capacity = 2 * Current Capacity.

(2) `Vector v = new Vector(int initialCapacity);`

(3) `Vector v = new Vector(int initialCapacity, int incrementalCapacity);`

(4) `Vector v = new Vector(Collection c);`

Vector Specific methods :- (10)

→ To add objects

① add(Object o) ⟶ C

② add(int index, Object o) ⟶ L

③ addElement(Object obj) ⟶ V

→ To remove Elements or objects

① remove(Object o) ⟶ C

② removeElement(Object o) ⟶ V

③ remove(int index) ⟶ L

④ removeElementAt(int index) ⟶ V

⑤ clear() ⟶ C

⑥ removeAllElements() ⟶ V

→ To retrive elements

① get(int index) ⟶ L

② elementAt(int index) ⟶ Y

③ firstElement(); ⟶ V

④ LastElement(); ⟶ V

→ Other methods

① int Size();

② int capacity();

* ③ Enumeration elements();

Eg:-
```
import java.util.*;
class VDemo1
{
 p.s.v.m(String[] args)
 {
  Vector V = new Vector();
  S.o.pln(V.capacity());
  for(int i=1 ; i<=10 ; i++)
  {
    V.addElement(i);
  }
  S.o.pln(V.capacity());
  V.addElement("A");
  S.o.pln(V.capacity());
  S.o.pln(V);
 }
}
```

O/P:-
```
10
10
20
[1, 2, 3, 4, 5, 6, ----10, A]
```

V.size() // 11 ← no. of objects

V.removeElement(9) // [1,2,3,4,5,6,7,8,10,A] ← object

V.removeElementAt(3) // [1,2,3,5,6,7,8,10,A]

V.removeAllElements() // []

(iv) Stack (c) :- (LIFO)

→ It is the child class of vector Contains only one Constructor

(i) Stack s = new Stack();

Methods :-

(i) Object push(Object o)

To insert an object into the Stack

(ii) Object pop();

To remove and returns top of Stack.

(iii) Object peek();

To return top of the Stack.

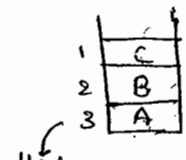
(iv) boolean empty();

returns true when Stack is Empty.

(v) int search(Object o)

returns the offset from top of the Stack if the Object is available, Otherwise returns -1.

Ex.



S.search("A"); 3
S.search("C"); 1
S.search("z"); -1

Ex.

```
import java.util.*;
class StackDemo
{
  p.s.v.m(String[] args)
  {
    Stack s = new Stack();
    S.push("A");
    S.push("B");
    S.push("C");
    S.o.pln(S); // [A B C]
    S.o.pln(S.search("A")); 3
    S.o.pln(S.search("z")); -1
  }
}
```

S.pop(); [A B

Cursors :-

Types of Cursors :-

→ If we want to get objects one by one from the Collection we should go for Cursor.

→ There are 3 types of Cursors available in java.

(i) Enumeration (1.0v)

(ii) Iterator (1.2v)

(iii) ListIterator (1.2v)

(i) Enumeration (1.0 Ver)

→ It is a Cursor to retrieve Objects one by one from the Collection.

→ It is applicable for legacy classes.

→ We can Create Enumeration object by using elements()

```
public Enumeration elements();
```

eg:-

```
Enumeration e = v.elements();
```

v → Vector object

→ Enumeration Interface defines the following 2 methods.

(i) public boolean hasMoreElements();

(ii) public Object nextElement();

Ex:-

```
import java.util.*;
Class EnumerationDemo
{
 p.s.v.m(String[] args)
 {
   Vector V = new Vector();
   for(int i=0 ; i<=10; i++)
   {
    V.addElement(i);
   }
   S.o.pln(v);   [0, 1, 2, 3 ----10]
   Enumeration e = V.elements();
   While(e.hasMoreElements())
   {
    Integer I = (Integer)e.nextElement();
    if(I%2 ==0)
    S.o.pln(I);
    }
    S.o.pln(V);    [0, 1, 2, 3, 4, --- 10]
  }
}
```

o/p:-

```
[0, 1, 2, 3 ----- 10]
0
2
4
6
8
10
[0, 1, 2, 3 ---- 10]
```

Limitations of Enumeration :-

→ Enumeration Concept is applicable only for Legacy classes & hence it is not a universal Cursor.

→ By using Enumeration we can get only ReadAccess & we can't perform any remove operations.

→ To over come these Limitations SUN people introduced Iterator in 1.2 version.

Iterator :-

→ We can apply Iterator Concept for any Collection objects. It is a universal Cursor.

→ While Iterating we can perform remove operation also, in addition to read operation.

→ We can get Iterator object by iterator() of Collection interface.

```
Iterator itr = c.iterator()
```

Any collection object

→ Iterator Interface defines the following 3 methods.

(i) public boolean hasNext();

(ii) public Object next();

(iii) public void remove();

Eg:-
```
import java.util.*;
class HashSetDemo
{
 public static void main(String[] args)
 {
   HashSet h = new HashSet();
    h.add("B");
    h.add("C");
    h.add("D");
    h.add("Z");
    h.add(null);
    h.add(10);
    S.o.pln(h.add("Z")); // false
    S.o.pln(h); // [null, D, B, C, 10, Z]
 }
}
```

O/P:- false
[null, D, B, C, 10, Z]

Note:- Insertion order is not preserved

(ii) LinkedHashSet (c):-

→ LinkedHashSet is the child class of HashSet.

→ It is exactly same as HashSet except the following differences.

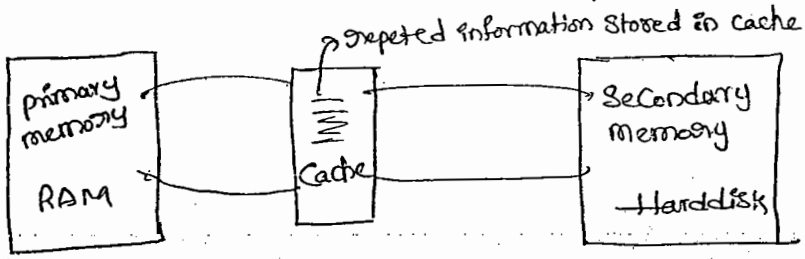
HashSet	LinkedHashSet
(i) The underlying D.S is Hashtable	i) The underlying D.S is a Combination of Hashtable & Linked List
(ii) Insertion order is not preserved	ii) Insertion order is preserved.
(iii) Introduced in 1.2V	iii) Introduced in 1.4V

→ In the above program if we are replacing HashSet with LinkedHashSet the following is the o/p.

o/p :- [B, C, D, Z, null, 10] i.e, insertion order is preserved.

Note :-

→ The main important application area of LinkedHashSet & LinkedHashMap is implementing Cache applications. where duplicates are not allowed & insertion order must be preserved.



3) SortedSet (I) :-

→ It is the child Interface of Set.

→ If we want to represent a group of individual objects according to some Sorting order. Then we should go for SortedSet

→ SortedSet Interface defines the following 6 Specific methods

(i) Object first()

→ returns the first element of SortedSet.

(ii) Object last()

→ returns last element of SortedSet

(iii) SortedSet headSet(Object obj)

→ returns the SortedSet whose elements are less than obj

(iv) SortedSet tailSet(Object obj)

→ Returns the SortedSet whose elements are greaterthan or equal to obj (≥)

(v) SortedSet subSet(Object obj1, object obj2)

→ Returns the SortedSet whose elements are ≥ obj1 but < obj2

(vi) Comparator comparator()

→ Returns Comparator object describes underlying Sorting technique

→ If we used default natural Sorting ordere then we will get null.

Ex:-

100
101
103
104
107
109

① first() ⟶ 100
② last() ⟶ 109
③ headSet(104) ⟶ [100, 101, 103]
④ tailSet(104) → [104, 107, 109]
⑤ subSet(101, 107) ⟶ [101, 103, 104]
⑥ Comparator() ⟶ null

Note:-

→ The default natural Sorting order for the no's is ascending order

→ The default natural Sorting order for characters & Strings are is alphabetical order (dictionary based Order).

TreeSet (c) :-

→ The underlying datastructure is Balanced Tree.

→ duplicate objects are not allowed.

→ Insertion order is not preserserved. because objects will be inserted according to some Sorting order.

→ Heterogeneous objects are not allowed. otherwise we will get "ClassCastException". & Null insertion is not possible ~~for the first object~~ Empty Set.

Constructors :-

(i) TreeSet t = new TreeSet();

→ Creates an Empty TreeSet object where the Sorting order is default natural Sorting order.

(ii) TreeSet t = new TreeSet(Comparator c)

→ Creates an Empty treeset object where the Sorting order is Customized Sorting order Specified by Comparator object.

(iii) TreeSet t = new TreeSet(Collection c)

(iv) TreeSet t = new TreeSet(SortedSet c)

Ex :-
```
import java.util.*;
class TreeSetDemo
{
  p.s.v.m(String[] args)
  {
```

```
TreeSet t = new TreeSet();
    t.add("A");
    t.add("a");
    t.add("B");
    t.add("z");
    t.add("L");
    //t.add(new Integer(10)); //CCE  ClassCastException
    //t.add(null); //--> NPE
    S.o.pln(t);  [A, B, z, L, a]
}
}
```

* Null acceptance :-

(i) For the Non-Empty TreeSet if we are trying to insert null we will get NullPointerException(NPE).

(ii) For the Empty TreeSet add the first element null insertion is always possible.

(iii) But after inserting that null, if we are trying to insert any-other, we will get NullPointerException (NPE).

```
eg:- import java.util.*;
     class TreeSetDemo1
     {
      p.S.v.m(String[] args)
      {
        TreeSet t = new TreeSet();
          t.add(new StringBuffer("A"));
          t.add(new StringBuffer("Z"));
          t.add(new StringBuffer("L"));
          t.add(new StringBuffer("B"));  o/p:- CCE
      }}  S.o.pln(t);
```

↳ If we are depending on default natural sorting order compolsary objects should be Homogeneous & Comparable otherwise we will get ClassCastException (CCE)

→ An object is said to be Comparable iff the corresponding class implements Comparable Interface.

↳ String class & all wrapper classes already implements Comparable Interface where as StringBuffer doesn't implements Comparable Interface. Hence, In the above Example we got ClassCastException.

Comparable Interface :-

→ This Interface present in java.lang package & Contains only one method i.e, "compareTo()".

```
public int compareTo(Object obj)

obj1.compareTo(obj2)
```

→ returns -ve iff obj1 has to come before obj2.

→ returns +ve iff obj1 has to come after obj2.

→ returns 0 iff obj1 & obj2 are equal (duplicate)

eg:-
```
import java.util.*;
class Test
{
  P.S.V.m(String[] args)
  {
    S.o.pln("A".compareTo("Z")); // -ve   -25
    S.o.pln("Z".compareTo("K")); // +ve   15
    S.o.pln("A".compareTo("A")); // 0     0
  }
}
```

→ When we are depending on default natural Sorting order internally JVM calls ~~Com~~ compareTo().

→ Based on the return-type JVM identifies the Location of the element in Sorting order.

obj1.compareTo(obj2)

- which object we are trying to add
- already existing object in TreeSet.

→
- → returns -ve iff obj1 has to come before obj2.
- → returns +ve iff obj1 has to come after obj2.
- → returns 0 iff obj1 & obj2 are equal

Eg:-

```
TreeSet t = new TreeSet();
t.add("z");
t.add("K");  ⟹ "K".compareTo("z"); -ve
t.add("D");  ⟹ "D".compareTo("K"); -ve
t.add("M");  ⟹ "M".compareTo("D") ⟹ +ve
t.add("D");      "M".compareTo("K") ⟹ +ve
                 "M".compareTo("z"); → -ve
//t.add(null);   "D".compareTo("D") ⟹ 0
S.o.println(t);
```

[0, K, M, z]

~~ClassCastException:NPE~~

null.compareTo("D") ⟹ RE ⟹ NPE

→ If we are not satisfied with default natural sorting order (or) if the default natural sorting order is not already available. Then we can define our own Customized sorting by using Comparator

> * Comparable ment for default natural sorting order.
> * Comparator ment for Customized sorting order.

Comparator (I):-

→ Comparator Interface present in java.util package & defines the following 2 methods.

① public int compare(Object obj1, object obj2):

- → returns -ve iff obj1 has to come before obj2
- → returns +ve iff obj1 has to come after obj2
- → returns 0 iff obj1 & obj2 are equal (duplicate).

obj1 ⇒ which object we are trying to add
obj2 ⇒ already existing object

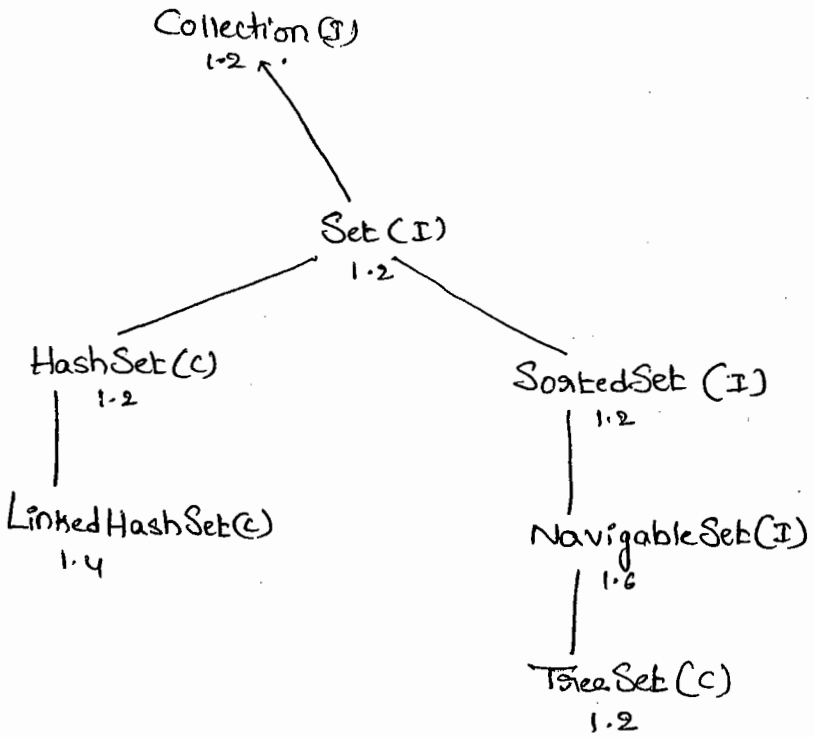
② public boolean equals(Object obj)

→ when ever we are implementing Comparator Interface compulsary we should provide implementation for compare(), 2nd method .equals() implentation is optional, because it is already available for our class from Object class through Inheritance.

(2) Set (I) :-

→ Set is Child Interface of Collection.

→ If we want to represent a group of objects where duplicates are not allowed & insertion order is not preserved, then we should go for Set.



→ Set Interface doesnot Contain any method we have to use only Collection Interface method.

(i) HashSet (C) :-

→ The underlying datastructure is Hashtable.

→ Duplicate objects are not allowed.

→ If we are trying to add duplicate objects we won't get any C.E or R.E add() simply returns false.

→ Insertion order is not preserved & all objects are inserted according to hashCode of the objects

→ Heterogeneous objects are allowed.

→ Null insertion is possible (only one) because duplicates are not allowed.

→ HashSet Implements Serializable & Clonable Interfaces.

* Constructors:-

1) HashSet h = new HashSet();

→ Creates an Empty HashSet object with default initial Capacity 16 & default fillRatio 0.75 (75%).

2) HashSet h = new HashSet(int initialCapacity);

→ Creates an Empty HashSet object with the specified initial Capacity & default fillRatio is 0.75.

3) HashSet h = new HashSet(int initialCapacity, float fillRatio);
(0 to 1)

4) HashSet h = new HashSet(Collection c);

fillRatio:-

→ After Completing (filling) the Specified ratio then only a new HashSet object will be Created that particular ratio is called fillRatio & load factor.

→ The default fillRatio is 0.75 but we can Customized this value.

Eg:-

```
import java.util.*;
class IteratorDemo
{
  public static void main(String[] args)
  {
    ArrayList l = new ArrayList();
    for(int i=0 ; i<=10 ; i++)
    {
      l.add(i);
    }
    S.o.pln(l);   [0,1,2,3,.....10]
    Iterator itr = l.iterator();
    while(itr.hasNext())
    {
      Integer I = (Integer)itr.next();
      if(I%2==0)
      {
        S.o.pln(I);      0
      }                  2
      else               4
      {                  6
        itr.remove();    8
      }                  10
    }
    S.o.pln(l);  [0,2,4,6,8,10]
```

Limitations of Iterator:-

(i) In the Case of Iterator & Enumeration we can always move towards the forward direction & we can't move backward direction. i.e these cursors are single directional cursors but not Bidirectional.

(ii) While performing Iteration we can perform only read & remove operations &

We can't perform replacement & addition of New objects.

→ To resolve these problem SUN people introduced ListIterator in 1.2 version.

i) ListIterator :-

→ ListIterator is the child Interface of Iterator.

→ While Iterating Objects by ListIterator we can move either to the forward or to the Backward direction. i.e ListIterator is a Bidirectional Cursor.

→ While Iterating By ListIterator we can perform replacement & addition of new objects also in addition to Read & Remove operations.

→ We can Create ListIterator object by using listIterator() of List Interface.

```
ListIterator litr = l.listIterator();
```

(l → any List object; listIterator → small letter)

→ ListIterator Interface defines the following 9 methods.

Forward:
(i) public boolean hasNext();
(ii) public Object next();
(iii) public int nextIndex();

Backward:
(iv) public boolean hasPrevious();
(v) public Object previous();
(vi) public int previousIndex();

⑦ public void remove();

⑧ public void set(Object new); → replace an object with new object

⑨ public void add(Object new); → add new obj.

Eg:-

```
import java.util.*;
class ListIteratorDemo
{
 Public static void main(String[] args)
 {
    LinkedList l = new LinkedList();
     l.add("balakrishna");
     l.add("venky");
     l.add("chiru");
     l.add("nag");
     S.o.pln(l);     [balakrishna, venky, chiru, nag]
     LinkedList
     ListIterator ltr = l.listIterator();
     while(ltr.hasNext())
     {
      String s = (String)ltr.next();
      if(s.equals("venki"))
      {
        ltr.remove();
      }
      if(s.equals("chiru"))
       {
         ltr.set("charan");
      }
      if(s.equals("nag"))
      {
        ltr.add("chittu")
      }}
      S.o.pln(l);          [Balakrisha, charan, nag, chittu]
 }}
```

Note:-

→ among 3 cursors ListIterator is the most powerfull Cursor, But it is applicable only for List objects.

Comparision table of 3-Cursors :-

Property	Enumeration (1.0v)	Iterator (1.2v)	ListIterator (1.2v)
① Is it legacy	yes	No	No
② It is applicable	only for Legacy classes	for any Collection objects	Only for List objects
③ movement	Single direction (only forward)	Single direction (forward)	bi-directional (forward & backward)
④ How to get it?	By using elements() method	By using iterator()	By using ListIterator()
⑤ Accessibility	only read	read & remove	read/remove/ replace/add
⑥ method	hasMoreElements() nextElement()	hasNext() next() remove()	9 methods

Eg:-
```
import java.util.*;
class TreeSetDemo3
{
 Public Static void main(String[] args)
 {
  TreeInteger I1 = (Integer) obj1;
      Integer I2 = (Integer
  TreeSet t = new TreeSet(new myComparator()); →①
  t.add(20);
  t.add(0); → Compare(0,20) → +ve
  t.add(15); → Compare(15,20) → +ve
             → Compare(15,0) → -ve
  t.add(5); → Compare(5,20) → +ve
            → Compare(5,0) → +ve
            → Compare(5,15) → -ve
  t.add(10); → Compare(10,20) +ve
             → Compare(10,0) +ve
             → Compare(10,15) +ve
             → Compare(10,5) -ve
  S.o.println(t);
 }
}
```
[20, 15, 10, 5, 0]

```
class MyComparator implements Comparator
{
  Public int Compare(Object obj1, Object obj2)
  {
    Integer I1 = (Integer) obj1;
    Integer I2 = (Integer) obj2;
    if(I1<I2)
      return +100;
    else if(I1>I2)
      return -100;
    else
      return 0;
  }
}
```
return((I1<I2)?+1:(I1>I2?-1:0));

→ If we are not passing Comparator object at line 1 (1) then JVM internally calls CompareTo() which is ment for default natural Sorting order. In this case the o/p is [0, 5, 10, 15, 20].

→ If we are passing comparator object at (1) then our own Compare method will be executed which is ment for Customized Sorting order. These case the o/p is [20, 15, 10, 5, 0]

Various alternatives of implementing compare():-

```
Class MyComparator implements Comparator
{
 public int Compare (Object obj1, Object obj2)
 {
   Integer I1 = (Integer) obj1;
   Integer I2 = (Integer) obj2;
 // return I1.CompareTo(I2); ⇒ [0, 5, 10, 15, 20]
 // return -I1.CompareTo(I2); ⇒ [20, 15, 10, 5, 0]
 // return I2.CompareTo(I1); ⇒ [20, 15, 10, 5, 0]
 // return -I2.CompareTo(I1); ⇒ [0, 5, 10, 15, 20]
 // return -1; ⇒ [10, 5, 15, 0, 20] ⇒ Reverse of insertion order
 // return +1; ⇒ [20, 0, 15, 5, 10] ⇒ insertion order.
 // return 0; ⇒ [20]
 }
}
```

(*) W.a.p to insert String objects into the TreeSet where the Sorting order is reverse of alphabetical order.

(A):

```
import java.util.*;

class TreeSetDemo2
{
  public static void m(String[] args)
  {
    TreeSet t = new TreeSet(new myComparator());
      t.add("A");
      t.add("Z");
      t.add("K");
      t.add("B");
      t.add("a");
    S.o.println(t);
  }
}

Class MyComparator implements Comparator
{
  public int compare(Object obj1, object obj2)
  {
    String s1 = (String)obj1;
    String s2 = obj2.toString();
    return -s1.compareTo(s2);
  }
}
```

Note:-
In object class Comparee method doesn't Contain Strings only contain object type so objects Can be Convert into Strings by using typecasting

Note:-
→ In objects & StringBuffer there is no CompareTo(), So we Can Convert into Strings.

* W.a.p to insert String & StringBuffer objects into the TreeSet where the Sorting order increasing length order. If two objects having the Same length then Consider their alphabetical order

```
import java.util.*;

Class TreeSetDemo12
{
  p.s.v.m(String[] args)
  {
    TreeSet t = new TreeSet(new MyComparator());
    t.add("A");
    t.add(new StringBuffer("ABC"));
    t.add(new StringBuffer("AA"));
    t.add("XX");
    t.add("ABCD");
    t.add("A");
    S.o.pln(t);    [A, AA, XX, ABC, ABCD]
  }
}

Class MyComparator implements Comparator
{
  public int Compare(Object obj1, Object obj2)
  {
    String S1 = obj1.toString();
    String S2 = obj2.toString();
    int l1 = S1.length();
    int l2 = S2.length();
    if(l1<l2)
      return -1;
    else(l2>l2)
      return +1;
    else
      return S1.CompareTo(S2);
  }
}
```

* W.a.p to insert StringBuffer objects into the TreeSet where the Sorting order alphabetical order?

```
import java.util.*;
Class TreeSetDemo10
{
  P.S.V.M(String[] args)
  {
    TreeSet t = new TreeSet(new MyComparator());
     t.add(new StringBuffer("A"));
     t.add(new StringBuffer("Z"));
     t.add(new StringBuffer("K"));
                              ("L")];
     S.o.println(t);   [A,K,L,Z]
  }
}
Class MyComparator implements Comparator
{
  Public int Compare(Object obj1, object obj2)
   {
      String s1 = obj1.toString();
      String s2 = obj2.toString();
      Return s1.CompareTo(s2);
   }
}
```

O/P :- [A, K, L, Z]

Note:-

So SB Can be Convert into String

→ In StringBuffer there is no CompareTo method

→ If we are depending on default natural Sorting order Compulsory objects should be Homogeneous & Comparable, Otherwise we will get CCE

→ If we are depending on our own Sorting by Comparator the objects need not be Comparable & Homogeneous.

Comparable Vs Comparator :-

① for predefined Comparable classes default natural Sorting order is already available if we are not satisfied with that we can define our own Customized Sorting By using Comparator

Ex:- String.

② for predefined NON-Comparable classes default natural Sorting order is not available Compulsary we should define Sorting by using Comparator object only.

Ex:- StringBuffer.

③ for our own Customized classes to define default natural Sorting order we can go for Comparable & to define Customized Sorting we should go for Comparator.

Ex:- Employee, Student, Customer.

```
import java.util.*;
class Employee implements Comparable
{
  int eid;
  Employee(int eid)
  {
    this.eid = eid;
  }
  public String toString()
  {
    return "E-" +eid;
  }
  public int CompareTo(Object obj)
  {
    int eid1 = this.eid;
    Employee e2 = (Employee)obj;
    int eid2 = e2.eid;
    if (eid1 < eid2)
      return -1;
    else if (eid1 > eid2)
      return +1;
    else
      return 0;
    }
  }
class CompCompDemo
{
  p s.v.m(String[] args)
{
```

```
Employee e1 = new Employee(200);
Employee e2 = new Employee(100);
Employee e3 = new Employee(500);
Employee e4 = new Employee(300);
Employee e5 = new Employee(700);
TreeSet t1 = new TreeSet();
t1.add(e1);
t2.add(e2);
t3.add(e3);
t4.add(e4);
t1.add(e5);
S.o.pln(t1);   [E-100, E-200, E-500, E-700]
TreeSet t2 = new TreeSet(new MyComparator());
t2.add(e1);
t2.add(e2);
t2.add(e3);
t2.add(e4);
t2.add(e5);
S.o.pln(t2);   [E-700, E-500, E-200, E-100]
}
}
Class MyComparator implements Comparator
{
  public int Compare(Object obj1, Object obj2)
  {
      Employee e1 = (Employee) obj1;
      Employee e2 = (Employee) obj2;
  } return e2.compareTo(e1);   // Reverse e1.compareTo(e2)
```

1), W.a.p to insert Employee objects into the TreeSet where default natural Sorting order is ascending order of Salaries. If Two Emp having the same Salary then Consider alphabetical orders of their names. &

* W.a. Comparator class to define Customized Sorting which is alphabetical order of Employee names. If two Employees having the same name then Consider descending order of their age.

* Comparision b/w Comparable & Comparator :-

Comparable	Comparator
1) We can use Comparable to define default natural Sorting order.	1) We can use Comparator to define Customized Sorting order.
2) This interface present in java.lang package.	2) This interface present in java.util package.
3) defines only one method i.e compareTo()	3) defines Two methods (i) compare() (ii) equals()
4) All wrapper classes & String class implements Comparable interface	4) No predefined class implements Comparator Interface.

Comparision table for Set implemented classes.?

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying D.S	Hashtable	Hashtable + Linked List	Balanced Tree
2) Insertion order	Not preserved	preserved	Not preserved
3) Sorting order	N.A	N.A	preserved
4) Heterogeneous Objects	allowed	allowed	Not allowed
5) Duplicate objects	not allowed	not allowed	not allowed
6) null acceptance	allowed (1)	allowed (1)	for the empty TreeSet add the first element null insertion is possible. in all other cases we will get NPE

15/6/25

Map (I) :-

→ If we want to represent a group of objects as key-value pairs Then we should go for Map. Both key & value are objects.

→ Both key & values are objects.

→ Duplicate keys are not allowed, But values can be duplicated.

→ Each key-value pair is Called Entry.

Ex:-

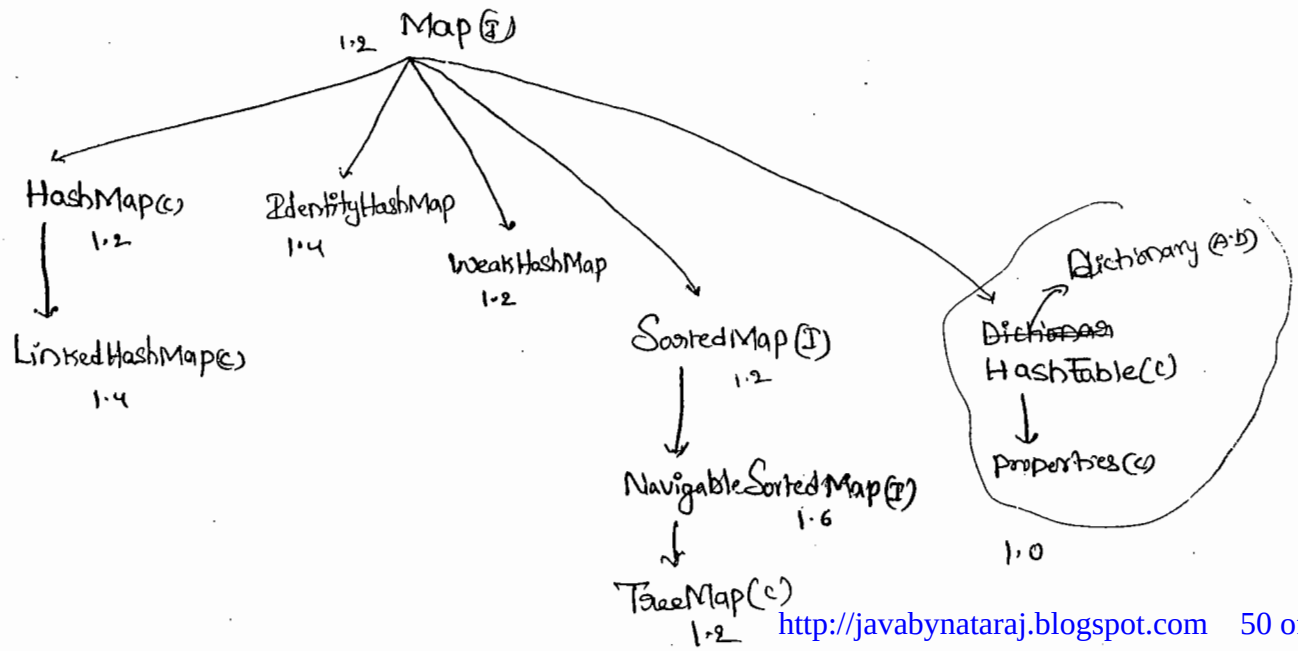
Rollno	Name
101	durga
102	Sunny
103	Ravi
104	Sambu
105	Sundar

key, entry, value

→ There is no relationship b/w Collection & Map.

→ *Collection ment for a group of individual objects where as *Map ment for a group of key-value pairs.

→ Map is not Child interface of Collection.



Methods of Map Interface :-

*① Object put(Object key, Object value);

→ To add key-value pair to the map

→ If the Specified key is already available then old value will be replaced with new value & old value will be returned.

② Void putAll(Map m)

→ To add a group of key-value pairs.

③ Object get(Object key)

→ Returns the value associated with Specified key

→ If the key is not available then we will get Null

④ Object remove(Object key);

⑤ boolean containsKey(Object key)

⑥ boolean containsValue(Object value)

⑦ int size();

⑧ boolean isEmpty();

⑨ Void clear()

① Set keySet();
② Collection values();
③ Set entrySet();

} Collection views of the Map.

Entry (Interface) :

→ Each key-value pair is called One Entry

→ Without existing Map Object there is no chance of Entry object Hence, Interface Entry is define inside Map Interfaces.

Code:

```
interface Map
{
  interface Entry
  {
    ①, Object getKey();
    ②, Object getValue();
    ③, Object setValue();
  }
}
```

(1) HashMap :-(C)

→ The underlying datastructure is HashTable

→ Heterogeneous objects are allowed for both keys & values.

→ duplicate keys are not allowed ~~for~~ but the values can be duplicated.

→ Insertion order is not preserved because it is based on HashCode of keys.

→ null key is allowed (only once)

→ null values are allowed (any number of times).

* differences b/w HashMap & HashTable :-

HashMap	HashTable
① No method is Synchronized	① Every method is Synchronized
② multiple Threads Can Operates Simultaneously & Hence HashMap object is not Thread Safe	② At a time only one Thread is allowed to Operate on HashTable Objects. Hence it is Thread Safe.
③ Threads are not required to wait & hence relatively performance is High.	③ It increases waiting time of the Thread & Hence performance is low.
④ null is allowed for both key & value	④ null is not allowed for Both key & values. otherwise we will get NPE
⑤ Introduced in 1.2 version & It is non-Legacy	⑤ Introduced, in 1.0 version & it is legacy

Q) How to get Synchronized version of HashMap?

A) → By default HashMap object is not Synchronized, but we can get Synchronized version by using SynchronizedMap() of Collections class.

```
Map M = Collections.synchronizedMap(HashMap hm);
```

Constructors :-

(i) HashMap m = new HashMap( );

→ Creates an EmptyHashMap object with default initial capacity level is '16' & default fillRatio 0.75 (75%).

(ii) HashMap m = new HashMap(int initialCapacity)

(iii) HashMap m = new HashMap(int initialCapacity, float fillRatio)

(iv) HashMap m = new HashMap(Map m)

Ex :-
```
import java.util.*;
class HashMapDemo
{
 P.S.V.m(String[] args)
 {
   HashMap m = new HashMap();
    m.put("chiranjeevi", 700);
    m.put("balaiah", 800);
    m.put("venkatesh", 1000);
    m.put("nagarjuna", 500);
    S.o.pln(m);  {venkatesh=1000, balaiah=800, chiranjeevi=700, nagarjuna=500}
    S.o.pln(m.put("chiranjeevi", 1000));  700
    Set s = m.keySet();
    S.o.pln(s);  [venkatesh, balaiah, chiranjeevi, nagarjuna]
    Collection c = m.values();
    S.o.pln(c);  [1000, 800, 500, 700].
     Set S1 = m.entrySet();
    Iterator its = S1.iterator();
```

```
while (its.hasNext())
{
  Map.Entry m1 = (Map.Entry) its.next();
  S.o.pln(m1.getKey() + "----" + m1.getValues());
  if (m1.getKey().equals("nagarjuna"))
    m1.setValue(10000);
}
S.o.pln(m);
}
}
```

nagarjuna 500
venkatesh 1000
balaiah 800
chiranjeevi 1000

→ {nagarjuna=10000, venkatesh=1000, balaiah=800, chiranjeevi=1000}

ii) LinkedHashMap :-

→ It is the child class of HashMap.

→ It is Exactly same as HashMap except the following differences

HashMap	LinkedHashMap
① The underlying D.S is HashTable	① The underlying D.S is HashTable + LinkedList
② Insertion Order is not preserved	② Insertion Order is preserved
③ Introduced in 1.2 version	③ Introduced in 1.4 Version

→ In the above program if we are replacing HashMap with LinkedHashMap, the following is the O/P.

{chiranjeevi=700, balaiah=800, venkatesh=1000, nagarjuna=500}

i.e insertion order is preserved

Note:-

→ The main application area of LinkedHashSet & LinkedHashMap are cache applications implementation where duplication is not allowed & insertion order must be preserved.

(iii) IdentityHashMap :-

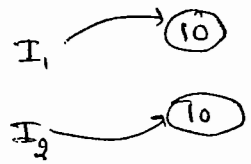
→ It is Exactly Same HashMap Except the following difference.

→ In the Case of HashMap to identify duplicate keys JVM always uses .equals(), which is mostly ment for Content Comparision.

→ If we want to use == operator instead of .equals() to identify duplicate keys we have to use IdentityHashMap. (== operator always ment for reference Comparision).

Eg:-

```
HashMap m = new HashMap();
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
m.put(i1, "pavan");
m.put(i2, "kalyan");
S.o.pln(m); {10=kalyan}
```



.equals() → Content
== → reference

I1 == I2 → false
I1.equals(I2) → True

→ In the above Code i1 & i2 are duplicate keys because i1.equals(i2) returns true.

→ If we replace HashMap with IdentityHashMap then the o/p is

{10=pavan, 10=kalyan}

→ i1 & i2 are not duplicate keys because i1==i2 returns false.

WeakHashMap :-

→ It is exactly same as HashMap except the following difference.

→ In the case of HashMap, Object is not eligible for g.c even though it doesn't have any external references if it is associated with HashMap. i.e, HashMap dominates Garbage Collector (g.c).

→ But in the case of WeakHashMap Even though object associated with WeakHashMap, it is eligible for g.c, if it doesnot have any external references. i.e G.c dominates WeakHashMap.

Eg:-

```
import java.util.*;
class WeakHashMapDemo
{
  p.s.v.m (String[] args) throws InterruptedException
  {
    HashMap m = new HashMap();
    Temp t = new Temp();
    m.put(t, "durga");
    S.o.pln(m);        {temp = durga}
    t = null;
    System.gc();
    Thread.sleep(5000);
    S.o.pln(m);
  }
}                      {temp = durga}
```

```
Class Temp
{
  public String toString()
  {
    return "temp";
  }
  public void finalize()
  {
    System.out.println("finalize method called");
  }
}
```

O/P:- {temp = durga}

{temp = durga}

→ If we replace HashMap with WeakHashMap then the o/p is

{temp = durga}

finalize method called

{}

(ii) SortedMap (I):-

→ If we want to represent a group of entries according to some sorting order then we should go for SortedMap. The sorting should be done based on the keys but not based on the values.

→ SortedMap interface is the child interface of Map.

→ SortedMap interface defines the following 6 specific methods

1. Object firstKey();
2. Object lastKey();
3. SortedMap headMap(Object key1);
4. SortedMap tailMap(Object key1);
5. SortedMap subMap(Object key1, Object key2)
6. Comparator comparator();

(iii) TreeMap (C):-

→ The underly D.S is RED-BLACK Tree,

→ Insertion order is not preserved & all entries are inserted according to some sorting order of keys.

→ If we are depending on default natural sorting order then the keys should be Homogeneous & Comparable. otherwise we will get ClassCastException (CCE).

→ If we are defining our own sorting order by Comparator then

The Keys need not be Homogeneous & Comparable.

→ There are no restrictions on values, they can be Heterogeneous & non-Comparable.

→ duplicate keys are not allowed but values can be duplicated.

Null acceptance:-

→ for the Empty TreeMap as the first Entry with null key is allowed. but after inserting that Entry if we are trying to insert any other Entry we will get NullPointerException (NPE).

→ for the NON-Empty TreeMap if we are trying to insert Entry with null key we will get NullPointerException (NPE)

→ There are no restrictions on null values. i.e, we can use null any no. of times any where for Map values.

Constructors:-

(i) TreeMap t = new TreeMap()

for default natural Sorting order.

(ii) TreeMap t = new TreeMap(Comparator c)

for Customized Sorting order.

(iii) TreeMap t = new TreeMap(Map m)

(iv) TreeMap t = new TreeMap(SortedMap m)

Eg:-

```
import java.util.*;
class TreeMapDemo3
{
  p.s.v.m(String[] args)
  {
    TreeMap m = new TreeMap();
    m.put(100, "zzz");
    m.put(103, "yyy");
    m.put(101, "xxx");
    m.put(104, 106);
    m.put(107, null);
    //m.put("FFFF", "xxx"); //CCE
    //m.put(null, "xxx"); //NPE
    S.o.println(m);
  }
}
```

{100=zzz, 101=xxx, 103=yyy, 104=106, 107=null}

O/P:-

{100=zzz, 101=xxx, 103=yyy, 104=106, 107=null}

eg:-

```
import java.util.*;

Class TreeMapDemo
{
  P.S.V.m(String[] args)
  {
     TreeMap t = new TreeMap(new MyComparator());
      t.put("xxx", 10);
      t.put("AAA",20);
      t.put("zzz", 30);
      t.put("LLL", 40);
      S.o.pln(t);
  }}

Class MyComparator implements Comparator
{
   public int compare(Object obj1, object obj2)
    {
        String S1 = Obj1.toString();
        String S2 = obj2.toString();
        return S2.compareTo(S1);
    }
}
```

o/p:- {zzz = 30, xxx = 10, LLL=40, AAA =20}

Hashtable©:

→ The Underlying datastructure is HashTable.

→ Heterogeneous objects are allowed for both keys & values

→ Insertion order is not preserved & it is based on HashCode of the keys.

→ null is not allowed for both key & values otherwise we will get NullPointerException (NPE).

→ duplicate keys are not allowed, but values can be duplicated.

→ All methods are Synchronized & Hence HashTable object is ThreadSafe.

Constructor:

(i) Hashtable h = new Hashtable()

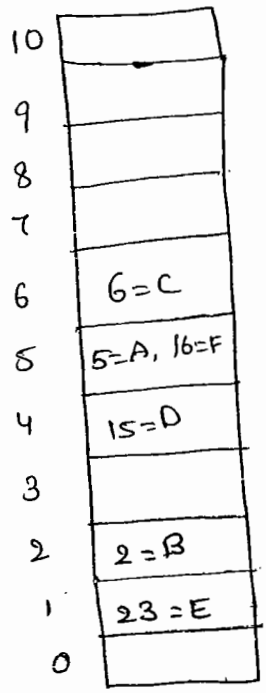
→ Creates an Empty Hashtable object with default initial capacity is "11" & default fillratio 75% (0.75).

(ii) Hashtable h = new Hashtable (int initialCapacity)

(iii) Hashtable h = new Hashtable (int initialCapacity, float fillratio)

(iv) Hashtable h = new Hashtable (Map m);

Eg:-
```
import java.util.*;
Class HashtableDemo
{
 P.S.V.m(String[] args)
 {
  Hashtable h = new Hashtable();
  h.put(new Temp(5), "A");
  h.put(new Temp(2), "B");
  h.put(new Temp(6), "C");
  h.put(new Temp(15), "D");
  h.put(new Temp(23), "E");
  h.put(new Temp(16), "F");
  //h.put("durga", null); //NPE
  System.out.println(h);
 }
}
Class Temp
{
 int i;
 Temp(int i)
 {
  this.i = i;
 }
 public int hashCode()
 {
  return i;
 }
 Public String toString()
 {
    return i+"";
 }}
```



{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}

from top to bottom & Right to Left

(iv) Properties(C):-

→ It is the child class of Hashtable

→ In our program if any thing which changes frequently (like database usernames, passwords, URL) never recommended to hardCode the value in the java program. Because for every change, we have recompile, rebuild, redeploye the application & Sometime even Sever restart also required. Which creates a big business impact to the client.

→ we have to Configuare those variables (~~properties~~ inside properties files & we have to read those values from javaCode.

→ the main advantage of this approach is, If any change in the properties file just redeployement is enough which is not a business impact to the client.

Constructor:-

(i) Properties p = new Properties();

→ In the Case of Properties both key & value should be String type

Methods:-

*(i) String getProperty(String PropertyName)

→ returns the value associated with Specified property.

(ii) String setProperty(String pname, String pvalue);

→ to Set a new property.

(iii) ~~String~~ Enumeration propertyNames();

* (iv) Void load(InputStream is)

→ To load the properties from properties files into java properties object.

(v) Void store(OutputStream os, String Comment)

→ To update properties from properties object into properties file.

Eg:-

```
import java.util.*;
import java.io.*;
class PropertiesDemo
{
 P.S.V.m(String[] args) throws IOException
 {
  Properties p = new Properties();
  FileInputStream fis = new FileInputStream("abc.properties");
  p.load(fis);
  System.out.println(p);
  String s = p.getProperty("venki");
  S.o.pln(s);
  p.setProperty("nag", "99999");
  FileOutputStream fos = new FileOutputStream("abc.properties");
  p.store(fos, "Updated by durga for SCJP Demo Class");
 }
}
```

abc.properties

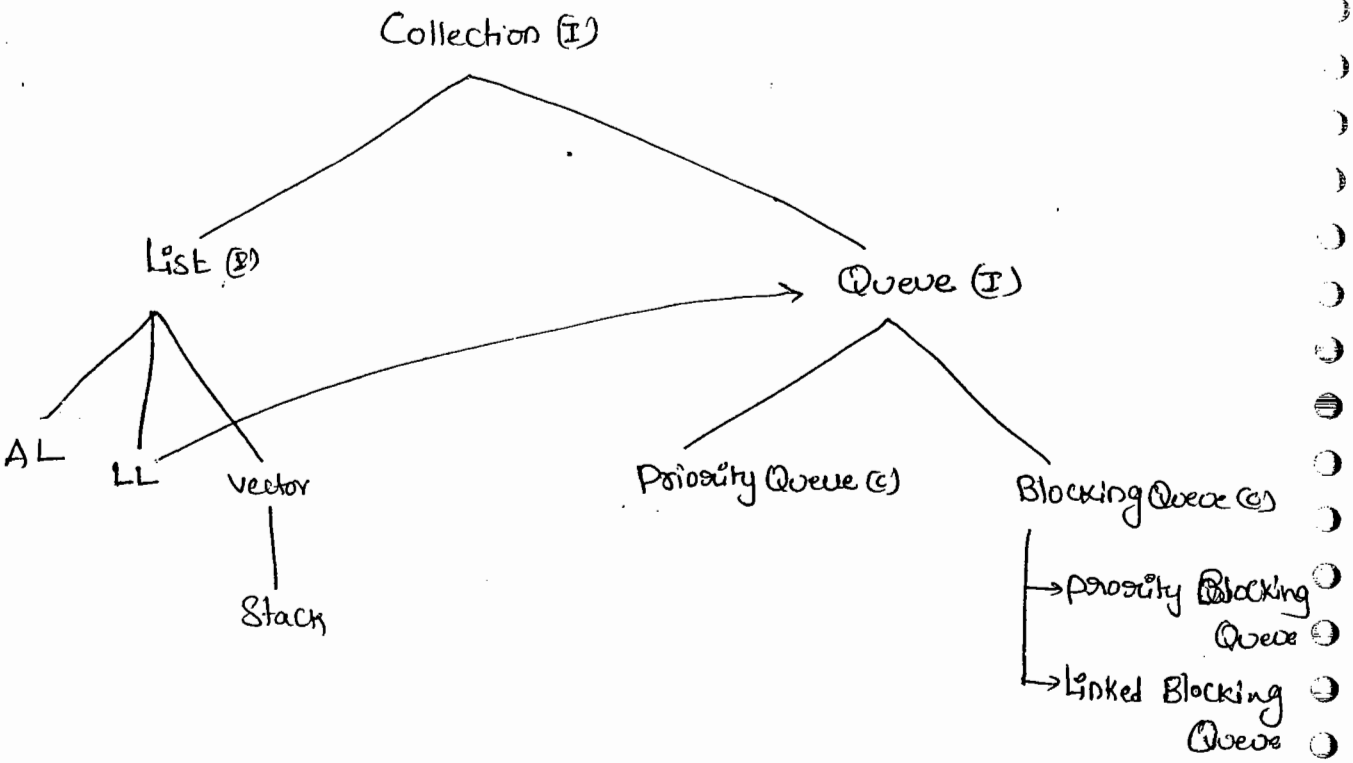
```
user = scott
venki = 8888
pwd = tiger
```

1.5 Version Enhancement :-

Queue(I) :-

→ It is the child Interface of Collection.

→ if we want to Represent a group of individual objects prior to processing then we should go for Queue.



→ Usually Queue follows FIFO (first in first out), But Based on our requirement we can change our order.

→ from 1.5 version onwards LinkedList implements Queue Interface.

→ LinkedList Based implementation of Queue always follows FIFO

Queue Interface methods :-

(i) boolean offer(Object obj)

→ To add an object into the Queue.

(ii) Object peek();

→ To return head element of the Queue. if Queue is Empty then this method returns null.

(iii) Object element();

→ To return head element of the Queue. if Queue is Empty then we will get RuntimeException saying NoSuchElementException

(iv) Object poll();

→ To remove & return head element of the Queue. If Queue is Empty then this method returns null.

(v) Object remove();

→ To remove & return head element of the Queue, if Queue is Empty then we will get RuntimeException saying NoSuchElementExcep

PriorityQueue(c) :-

→ This is the DataStructure to hold a group of individual Objects prior to processing According to Some priority.

→ The priority can be either default natural Sorting order or Customized Sorting order.

→ if we are depending on default natural Sorting Compulsary objects should be Homogeneous & Comparable otherwise we will get ClassCastException.

→ if we are defining our own Customized Sorting by Comparator Then the objects need not be Homogeneous & Comparable.

→ duplicate objects are not allowed.

→ Insertion order is not preserved.

→ null insertion is not possible even as first element also.

Constructors :-

(i) PriorityQueue q = new PriorityQueue();

→ Creates an Empty PriorityQueue with default initialCapacity "11" & priority order is default natural Sorting order.

(ii) PriorityQueue q = new PriorityQueue(int initialCapacity);

(iii) PriorityQueue q = new PriorityQueue(int initialCapacity, Comparator c);

(iv) PriorityQueue q = new PriorityQueue(Collection c);

(v) PriorityQueue q = new PriorityQueue(SortedSet s)

Eg:-

```
import java.util.*;
class PriorityQueueDemo
{
  p.s.v.m(String[] args)
  {
    PriorityQueue q = new PriorityQueue();
    S.o.pln(q.peek()); //null
    // S.o.pln(q.element()); //NSE  NoSuchElementException
    for(int i=0 ; i<=10 ; i++)
    {
      q.offer(i);
    }
    S.o.pln(q); // [0,1, 2, 3, 4, 5, ----10]
    S.o.pln(q.poll()); //0
    S.o.pln(q); // [1, 2, 3, 4, 5 ---10]
  }
}
```

eg 2:-

```
import java.util.*;
class priorityQueueDemo2
{
  p.s.v.m (String[] args)
  {
```

```
PriorityQueue q = new PriorityQueue(15, new MyComparator());
    q.offer("A");
    q.offer("Z");
    q.offer("L");
    q.offer("B");
    S.o.println(q); // [Z, L, B, A]
  }
}

Class MyComparator Implements Comparator
{
  Public int Compare(Object obj1, Object obj2)
  {
    String S1 = (String) obj1;
    String S2 = Obj2.toString();
    return S2.compareTo(S1);
  }
}
```

o/p:. [Z, L, B, A]

1.6 Version Enhancements :-

(i) NavigableSet (I) :-

→ It is the child Interface of SortedSet.

→ This Interface defines several methods to provide Support for Navigation for the TreeSet object.

→ The following List of various methods present in NavigableSet.

(i) Ceiling(e) :-

→ returns the lowest element which is >= e

(ii) higher(e) :-

→ returns the lowest element which is > e

(iii) floor(e) :-

→ returns highest element which is <= e.

(iv) lower(e) :-

→ returns the highest element which is < e.

(v) pollFirst() :-

→ remove & returns first element

(vi) pollLast() :-

→ remove & returns last element.

(vii) desendingSet() :-

→ returns the NavigableSet in reverse order.

Eg:

```
import java.util.*;
class NavigableSetDemo
{
 P.S.V.m(String[] args)
 {
   TreeSet<Integer> t = new TreeSet<Integer>();
   t.add(1000);
   t.add(2000);
   t.add(3000);
   t.add(4000);
   t.add(5000);
   S.o.pln(t) ; [1000, 2000,
   S.o.pln(t.ceiling(2000)); 2000
   S.o.pln(t.higher(2000)); 3000
   S.o.pln(t.floor(3000)); 3000
   S.o.pln(t.lower(3000)); 2000
   S.o.pln(t.pollFirst()); 1000
   S.o.pln(t.pollLast()); 5000
   S.o.pln(t.descendingSet()); [4000, 3000, 2000]
   S.o.pln(t); [2000, 3000, 4000]
 }
}
```

(ii) NavigableMap (I):-

→ It is the child interface of SortedMap to define several method for Navigation purposes.

→ The following is the list of methods present in NavigableMap.

(i) CeilingKey(e)

(ii) higherKey(e)

(iii) ~~floor~~FloorKey(e)

(iv) lowerKey(e)

(v) pollFirstEntry()

(vi) pollLastEntry()

(vii) descendingMap()

Eg:-

```
import java.util.*;
class NavigableMapDemo
{
  p.s.v.m(String[] args)
  {
    TreeMap<String, String> t = new TreeMap<String, String>();
    t.put("b", "banana");
    t.put("c", "cat");
    t.put("a", "apple");
    t.put("d", "dog");
    t.put("g", gun");
    S.o.println(t); // a=apple, b=banana, c=cat, d=dog, g=gun
```

```
S.o.pln(t.ceilingKey("c"));    c
S.o.pln(t.higherKey("e"));     g
S.o.pln(t.floorKey("e"));      d
S.o.pln(t.lowerKey("e"));      d
S.o.pln(t.pollFirstEntry());   a = apple
S.o.pln(t.pollLastEntry());    g = gun
S.o.pln(t.desendingMap());     {d = dog , c = cat , b = banana}
S.o.pln(t);                    {b = banana , c = cat , d = dog}
}
}
```

Collections class

Collections class:-

→ It is an utility class present in java.util package

→ It defines several utility methods for Collection implemented class objects

Sorting the elements of a list :-

→ Collections class defines the following methods to sort elements of a List.

① public static void sort(List l) :-

→ We can use these method to sort according to natural sorting order.

→ In this case comparing elements should be Homogeneous & Comparable. Otherwise we will get ClassCastException.

→ List should not contain null, otherwise we will get NullPointerException

② public static void sort(List l, Comparator c) :-

→ To sort elements of a List according to Customized sorting order

Searching the elements of a List :-

→ Collections class defines the following method to search elements of a List

① public static int binarySearch(List l, Object obj)

→ If the List is sorted according to natural sorting order then we have to use this method.

⑨ public static int binarySearch(List l, Object key, Comparator c)

→ If the List is Sorted according to Comparator Then we have to use This method.

Conclusion :-

→ Internally binarySearch method uses BinarySearch algorithm.

→ Before Calling binarySearch() method Compulsary the List should be Sorted. otherwise we will get unpredictable results.

→ Successfull Search returns index.

→ unsuccessfull Search returns insertion point

→ Insertion point is the Location where we can place element in the Sorted List.

→ If the List is Sorted according to Comparator Then at the time of Search also we should pass the Same Comparator Otherwise we will get unpredictable results.

Ex:- To Search elements of list

```
import java.util.*;
class CollectionsSearchDemo
{
   p.s.v.m(String[] args)
   {
      ArrayList l = new ArrayList();
         l.add("z");
         l.add("A");
         l.add("m");
```

```
l.add("K");
l.add("a");
S.o.pln(l);   [Z, A, M, K, a]
Collections.sort(l);
```

-1	-2	-3	-4	-5	-6
A	K	M	Z	a	
0	1	2	3	4	

```
S.o.pln(l);
```

A	K	M	Z	a

```
S.o.pln(Collections.binarySearch(l,"z"));  3
S.o.pln(Collections.binarySearch(l,"j"));  -2
}
}
```

Ex2:-

```
import java.util.*;
class CollectionsSearchDemo1
{
  p.s.v.m(——)
  {
    ArrayList l = new ArrayList();
    l.add(15);
    l.add(0);
    l.add(20);
    l.add(10);
    l.add(5);
    S.o.pln(l);
```

15	0	20	10	5

```
    Collections.sort(l, new MyComparator());
    S.o.pln(l);
```

20	15	10	5	0

```
    S.o.pln(Collections.binarySearch(l,10,new MyComparator()));  //2
    S.o.pln(Collections.binarySearch(l,13,new MyComparator()));  //-3
    S.o.pln(Collections.binarySearch(l,17));  // -6 unpredictable
  }
}
```

-1	-2	-3	-4	-5	-6
20	15	10	5	0	
0	1	2	3	4	

because it is not passing Comparator()

```
Class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2)
    {
        Integer i1 = (Integer) obj1;
        Integer i2 = (Integer) obj2;
        return i2.compareTo(i1);
    }
}
```

Note :-

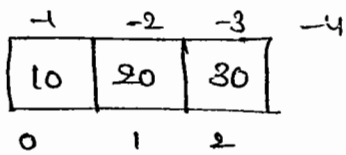
→ For the List Contains n elements Range of Successfull Search

① Range of Successfull Search : 0 to n-1

② Range of unsuccessfull Search : $-(n+1)$ to -1

③ total Range : $-(n+1)$ to $n-1$

Ex:-



Range of Successful Search : 0 to 2

Range of unsuccessful Search : -4 to -1

Total Range : -4 to 2

Reversing the elements of a List :-

→ Collections class defines the following reverse method for this

```
Public Static void reverse(List l);
```

Ex:- To Reverse elements of List

```
import java.util.*;
Class CollectionsReverseDemo
{
   P.S.V.m(_____)
   {
      AL l = new AL();
      l.add(15);
      l.add(0);
      l.add(20);
      l.add(10);
      l.add(5);
     S.o.pln(l);
```

15	0	20	10	5

```
     Collections.reverse(l);
     S.o.pln(l);
```

5	10	20	0	15

```
   }
}
```

reverse() Vs reverseOrder() :-

→ We can use reverse() method to reverse the elements of a List and this method Contain List Argument

→ Collections class defines reverse order method also to return Comparator object for reversing original Sorting order

Comparator c1 = Collections.reverseOrder(Comparator c)

- c1 → decending order
- c → asending order

→ reverseOrder() method ~~Can take~~ Contains Comparator argument whereas reverse() Contains List arguments.

Ex :- TO REVERSE ELEMENTS OF LIST

```
import java.util.*;
class CollectionsReverseDemo
{
  public static void main(String[] args){
      ArrayList l = new ArrayList();
      l.add(15);
      l.add(0);
      l.add(20);
      l.add(10);
      l.add(5);
      S.o.pln(l);
      Collections.reverse(l);
  }  S.o.pln(l);
```

S.o.pln(l); → | 15 | 0 | 20 | 10 | 5 |

S.o.pln(l); → | 5 | 10 | 20 | 0 | 15 |

Arrays class

Arrays class :-

→ It is an utility class present in util package, To define Several utility methods for Arrays for both primitive Arrays & Object type Arrays.

Sorting the elements of Array :-

→ Arrays class defines the following methods for this.

① public static void Sort(primitive[] p);

→ To Sort elements of primitive Array According to natural Sorting order

② public static void Sort(Object[] a)

→ To Sort elements of Object Array According to natural Sorting order.

→ In this Case Compulsary the elements should be Homogeneous & Comparable. Otherwise we will get ClassCastException.

③ public static void Sort(Object[] a, Comparator c) :-

→ To Sort elements of Object[] according to Customized Sorting order.

Note :-

Primitive Arrays Can be Sorted only by natural Sorting order where as Object Arrays Can be Sorted either by natural Sorting order or by Customized Sorting order.

Ex!- To SORT elements of Arrays

ArraysSortDemo.java

```
import java.util.Arrays;
import java.util.Comparator;

Class ArraysSortDemo
{
  public static void main(String[] args)
  {
    int[] a = {10, 5, 20, 11, 6};

    S.o.pln("primitive Array before Sorting:");

    for(int a1 : a)
    {
      S.o.pln(a1);          10
    }                       5
                            20
                            11
                            6
    Arrays.sort(a);

    S.o.pln("primitive Array After Sorting:");

    for(int a1 : a)
    {
      S.o.pln(a1);          5
                            10
                            11
                            20
    }

    String[] s = {"A", "Z", "B"};

    S.o.pln("object Array Before Sorting:");

    for(String a2 : s)
    {
      S.o.pln(a2);          A
                            Z
                            B
    }
    Arrays.sort(s);

    S.o.pln("object Array After Sorting:");
```

```
for (String a1 : s)
{
  S.o.pln(a1);   A B Z
}
Arrays.sort(s, new MyComparator());
S.o.pln("Object Array After Sorting by Comparator:");
for (String a1 : s)
{
  S.o.pln(a1);   Z B A
}}
Class MyComparator implements Comparator {
  public int compare(Object o1, Object o2) {
      String s1 = o1.toString();
      String s2 = o2.toString();
      return s2.compareTo(s1);
}}
```

Searching the elements of Array :-

→ Arrays class defines the following search methods for this.

① public static int binarySearch(primitive[] p, primitive key)

② public static int binarySearch(Object[] o, Object key)

③ public static int binarySearch(Object[] o, Object key, Comparator c)

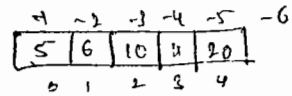
Note:-

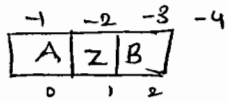
All rules of these binarySearch() method are Exactly same as Collections class binarySearch() method.

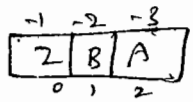
Ex:-

```
import java.util.*;
import static java.util.Arrays.*;
class ArraysSearchDemo
{
 P.S.V.m(——)
 {
  int[] a = {10, 5, 20, 11, 6};
  Arrays.sort(a); // Sort by natural order
  S.o.pln(Arrays.binarySearch(a,6)); //1
  S.o.pln(Arrays.binarySearch(a,14)); //-5

  String[] s = {"A", "Z", "B"};
  Arrays.sort(s);
  System.out.println(binarySearch(s,"Z")); //2
  S.o.pln(binarySearch(s,"S")); //-3
  Arrays.sort(s, new MyComparator());
  S.o.pln(binarySearch(s,"Z", new MyComparator())); //0
  S.o.pln(binarySearch(s,"S", new MyComparator())); //-2
  S.o.pln(binarySearch(s,"N")); //unpredictable result
 }
}
class MyComparator implements Comparator
{
  public int Compare(Object o1, Object o2)
  {
```







```
        String S1 = O1.toString();
        String S2 = O2.toString();
        return S2.compareTo(S1);
    }
}
```

Converting Arrays to List :-

① public static List asList(Object[] a)

→ By using this method we are not creating an independent List object just we are creating List view for the existing Array Object.

→ By using List reference if we perform any operation the changes will be reflected to the Array reference. Similarly, By using Array reference if we perform any changes those changes will be reflect to the List.

→ By using List reference we can't perform any operation which varies the size, (i.e, add & remove) otherwise we will get RuntimeException saying "UnSupported-Operation-Exception" (UOE).

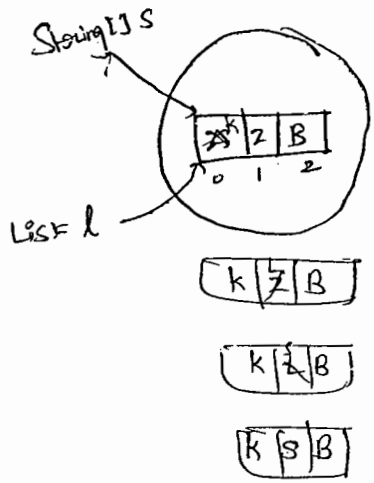
→ By using List reference we can perform replacement operation But replacement should be with the same-type of element only otherwise we will get RuntimeException saying "ArrayStoreException"

Eg:- To view Array IN LIST FORM.

ArrayAsListDemo.java ,

```
import java.util.*;

class ArraysAsListDemo
{
  public static void main(String[] args)
  {
    String[] s = {"A", "Z", "B"};
    List l = Arrays.asList(s);
    S.o.pln(l); // [A, Z, B]
    S[0] = "K";          [A, Z, B]  [K, Z, B]
    S.o.pln(l); // [K, Z, B]
    l.set(1, "L");       [K, Z, B]
    for(String s1 : s)
    S.o.pln(s1); //[K, L, B]
    l.add("durga"); R.E // UOE
    l.remove(2); R.E // UOE
    l.set(1, "S");       [K, Z, B] => [K, S, B]
    l.set(1, 10); R.E // ArrayStoreException
  }
}
```



01/05/11

Generics (1.5v)

1) Introduction
2) Generic classes
3) Bounded types
4) Generic methods
5) Wild Card Character
6) Communication with non-Generic Code.
7) Conclusions.

Introduction :-

→ Arrays are always Safe w.r.t type.

→ for Example, if our programme requirement is to add only String objects then we can go for String[] array. for this array we can add only String type of objects, by mistake if we are taying to add any other type we will get Compiletime-Error.

Ex :- String[] s = new String[600];

s[0] = "dorga"; ✓

s[1] = "pavan"; ✓

s[2] = new Student[]; ✗

Type-Safe.

C.E :- incompatable types
found : Student
required : String.

→ Hence In the Case of Arrays we can always give the guarantee about the the Type of elements. String[] array Contains only String objects, (i.e Strings) due to this arrays are always Safe to use w.r.t type.

→ But Collections are not Safe to use w.r.t type. for Example if our programme requirement is to hold only String objects & if we are using ArrayList, By mistake if we are trying to add any other type to the List we won't get any Compiletime-Error But program may fail at Runtime.

Eg:

```
ArrayList l = new ArrayList();
l.add("durga");
l.add("Sainu");
l.add(new Student());
:
String name1 = (String)l.get(0);
String name2 = (String)l.get(1);
String name3 = (String)l.get(2);
```

R.E:. ClassCastException.

→ There is no guarantee that Collection can Hold a particular type of objects. Hence w.r.t type Collections are not Safe to use.

Case2 :-

→ In the Case of Arrays at the time of retrieval it is not required to perform any TypeCasting.

Ex:-

```
String[] s = new String[600];
s[0] = "durga";
:
String name1 = s[0];
```

TypeCasting is not required.

→ But in the Case of Collections at the time of retrival compulsary we should perform TypeCasting otherwise we will get CompileTimeError.

Ex:-

```
ArrayList l = new ArrayList();
l.add("durga");
:
String name1 = l.get(0);
```

C.E:- Incompatable types
found : object
required : String

But

```
String name1 = (String)l.get(0);
```

→ Hence, in the Case of Collections TypeCasting is mandatory which is a bigger headache to the programmer.

→ To overcome the above problems of Collections (TypeSafe & TypeCasting) Sun people introduced Generics Concepts in 1.5Version. Hence the main objectives of Generic Concepts are,

Hence the main objectives of Generics Concepts are,

1) To provide Type safety to the Collections. So that they can hold only a particular Type of objects.

2) To resolve Type Casting problems.

→ for Example — to Hold only String Type of objects a Generic version of ArrayList we can declare as follows.

parameter-type.

Basetype → `ArrayList<String> l = new ArrayList<String>();`

→ for this ArrayList we can add only String type of Objects, by mistake if we are trying to add any other type we will get Compiletime Error. i.e, we are geting Type-Safety.

```
l.add("durga");  ✓
l.add("Sunny");  ✓
l.add("10");     ✓
l.add(10);       ✗  C.E:- Cannot find Symbol
                        Symbol : method add(int)
                        location : class ArrayList<String>
```

→ At the time of retrival it is not required to perform any Type Casting.

```
String name1 = l.get(0);  ✓
```

↙

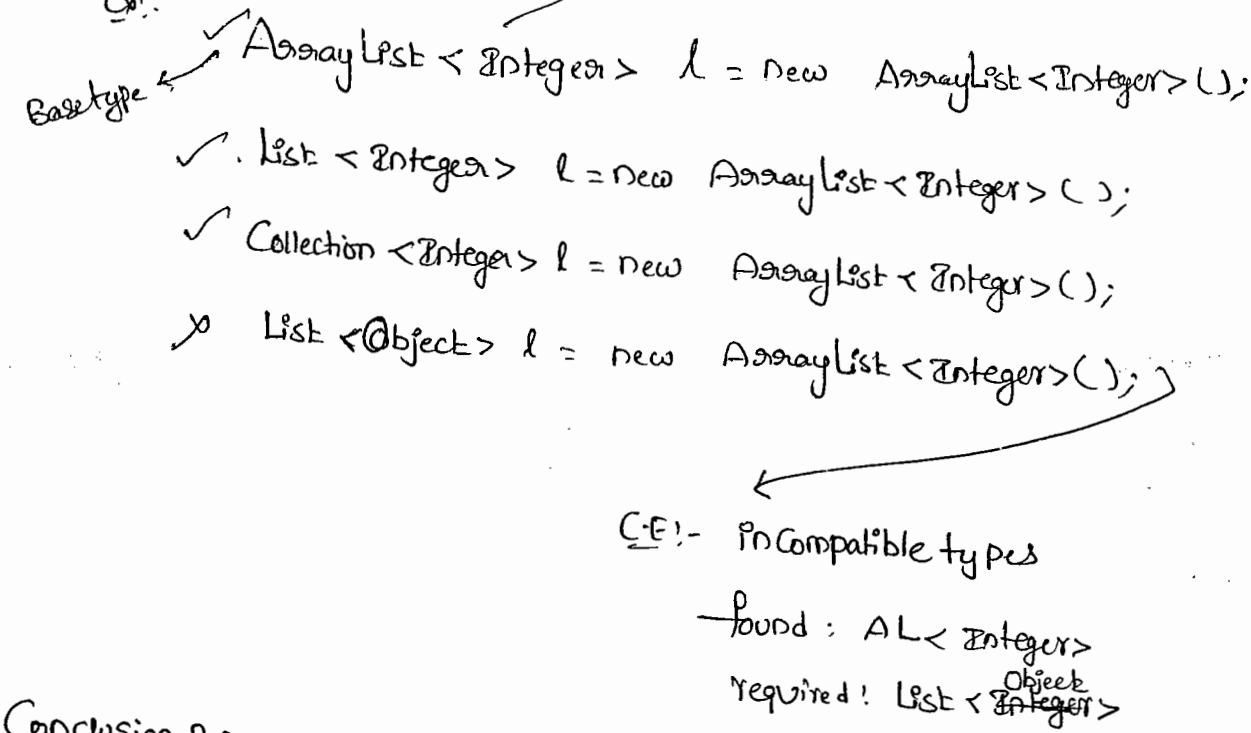
Type Casting is not required

Conclusion 1 :-

→ Usage of parent class reference to hold child class objects is considered as polymorphism.

→ polymorphism concept is applicable only for base type. but not for parameter type.

Ex:-



Conclusion 2 :-

→ for the parameter-type we can use any class or interface name & we can't use primitive type. vilation leads to CompileTime Error.

Ex:- ArrayList<int> l = new ArrayList<int>();

C.E:-
unexpected type
found : int
required : reference

C.E:-
Unexpected type
found : int
required : reference

Generic - classes :-

→ Until 1.4V a non-Generic Version of ArrayList class is declared as follows.

```
class ArrayList
{
    add(Object o);
    Object get(int index)
}
```

→ The argument to the add() method is Object. Hence we can add any type of object due to this we are not getting Type-Safety.

→ The return type of get() method is Object, Hence At the time of retrieval Compulsary we should perform Type Casting.

→ But in 1.5V a Generic Version of ArrayList class is declared as follows.

```
class ArrayList<T>        // T → Type parameter.
{
    add<T t>
    T get(int index)
}
```

→ Based on our runtime requirement Type parameter 'T' will be replaced with Corresponding provided Type.

→ for Example, To hold only String type of object we have to Create Generic Version of ArrayList Object as follows.

```
ArrayList<String> l = new ArrayList<String>();
```

→ for this requirement the corresponding loaded version of ArrayList Class is,

```
Class ArrayList<String>
{
    add(String s)
    String get(int index)
}
```

→ add() method can take String as the argument hence we can add only String type of objects. By mistake if we are trying to add any other type we will get CompileTimeError. i.e, we are getting Type-Safety.

→ The return type of get() method is String, Hence at the time of retrival we can assign directly to the String type variable it is not required to perform any TypeCasting.

Note :-

1) As the Type parameter we can use any valid java identifiers but it is Convension to use "T".

Eg:- ✓ Class AL<X>
{
}

Class AL<Durga>
{
}

2) We can pass any no. of type parameters but it need not be one. class

Ex:-

```
class HashMap<K,V>
{
}
```

```
HashMap<String, Integer> m = new HashMap<String, Integer>();
```

Key type

Value type

→ Through Generics we are associating a type-Parameter to the classes. Such type of parameterized classes are called Generic-classes.

→ We can define our own Generic classes also.

Ex:-

```
class Gen<T>
{
  T ob;
  Gen(T ob)
  {
    this.ob = ob;
  }
  public void show()
  {
    S.o.println("The Type of ob is :" +ob.getClass().getName());
  }
  public T getOb()
  {
    return ob;
  }
```

```
Class GenDemo
{
  p.s.v.m(String[] args)
  {
    Gen<String> g1 = new Gen<String>("durga");
    g1.show();  // the type of ob is : java.lang.String
    S.o.pln(g1.getOb());  durga

    Gen<Integer> g2 = new Gen<Integer>(10);
    g1.show();  // the type of ob is : java.lang.Integer.
    S.o.pln(g2.getOb());  10
  }
}
```

① ②

Bounded Types :-

→ We can Bound the type parameter for a particular range By using extends keyword.

ex(1) :-

```
Class Test<T>
{
}
```

→ As the type parameter we can pass any type hence it is UnBounded type.

✓ `Test<String> t1 = new Test<String>();`

✓ `Test<Integer> t2 = new Test<Integer>();`

Eg:- Class Test<T extends Number>
{
}

→ As the type parameter we can pass either Number type or its child classes. it is bounded type.

✓ Test<Integer> t1 = new Test<Integer>();

✗ Test<String> t2 = new Test<String>();

C.E:- Type parameter java.lang.String is not with in its bound

→ We can't Bound Type Parameter By using implements & Super Keywords :

Ex:- ① Class Test<T implements Runnable>
{
}

② Class Test<T Super Integer>
{
}

But,

→ implements Keyword purpose we can Survive by using extends Keyword only

Ex:- Class Test<T extends X>
{
}
↳ class / interface.

→ X → Can be either class/interface.

→ if X is a class then as the Type parameter we Can provide either X type or it's child classes.

→ if X is an interface as the Type parameter we Can provide either X type or it's implementation classes.

ex!-

```
Class Test<T extends Runnable>
{
}
```

✓ Test<Runnable> t1 = new Test<Runnable>();

✓ Test<Thread> t2 = new Test<Thread>();

✗ Test<String> t = new Test<String>();

C.E!- Type parameter java.lang.String is not within its Bound

→ We Can Bound the Type parameter even in Combination also.

ex!-

Class Test<T extends Number & Runnable>

→ As the Type parameters we Can pass any type which is the child Class of Number & implements Runnable interface.

ex!-

✓ ① Class Test<T extends Runnable & Comparable)

✓ ② class Test<T extends Number & Runnable & Comparable)

✗ ③ Class Test<T extends Number & Thread>

→ We Can't extend more than one class at a time.

✗ ⑤ Class Test<T extends Runnable & Number>

→ we have to take first class & then interface.

Generic Methods & Wild Card Character ?

→ ① m1(ArrayList<String> l) ✓

→ this method is applicable for ArrayList<String> (ArrayList of only String type).

→ within the method we can add String type objects & null to the List if we are trying to add any other type we will get Compiletime Error.

eg:- m1(ArrayList<String> l)
{
l.add("A"); ✓
l.add(null); ✓
l.add(10); ✗

② m1(ArrayList<? extends X> l) ✓

→ we can call this method by passing ArrayList of any type. But within the method we can't add any-type except null to the list. Because we don't know the type Exactly.

③ Eg:-
m1(ArrayList<?> l)
{
l.add(null); ✓
l.add("A"); ✗
l.add(10); ✗
}

3) m1(AL<? extends X> l) ✓

→ If X is a class then we can call this method by passing ArrayList of either X type or its child classes.

→ If X is an interface then we can call this method by passing ArrayList of either X type or its implementation class

→ In this case also we can't add any type of elements to the list Except null

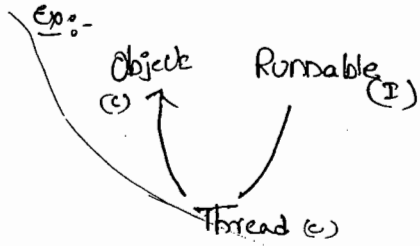
4) m1(ArrayList<? super X> l) ✓

→ If X is a class then this method is applicable for ArrayList of either X type or its Super classes.

→ If X is an interface then this method is applicable for ArrayList of either X type or Super classes of implementation class of X

→ With in the method we can add only X type objects & null to the List

Ex:-



Q) Which of the following declarations are valid?

① AL<String> l = new AL<String>(); ✓

② AL<?> l = new AL<String>(); ✓

③ AL<E, extends String> l = new AL<String>(); ✓

④ AL<? super String> l = new AL<String>(); ✓

⑤ AL<? extends Object> l = new AL<String>(); ✓

✓ ⑥ AL < ? extends Number > l = new AL < Integer > ();

✗ ⑦ AL < ? extends Number > l = new AL < String > ();

C.E:- Incompatible types

found : AL<String>

required: AL < ? extends Number >

✗ ⑧ AL < ? > l = new AL < ? extends Number > ();

✗ ⑨ AL < ? > l = new AL < ? > ();

C.E:- unexpected type

found : ?

required : Class or interface without bounds.

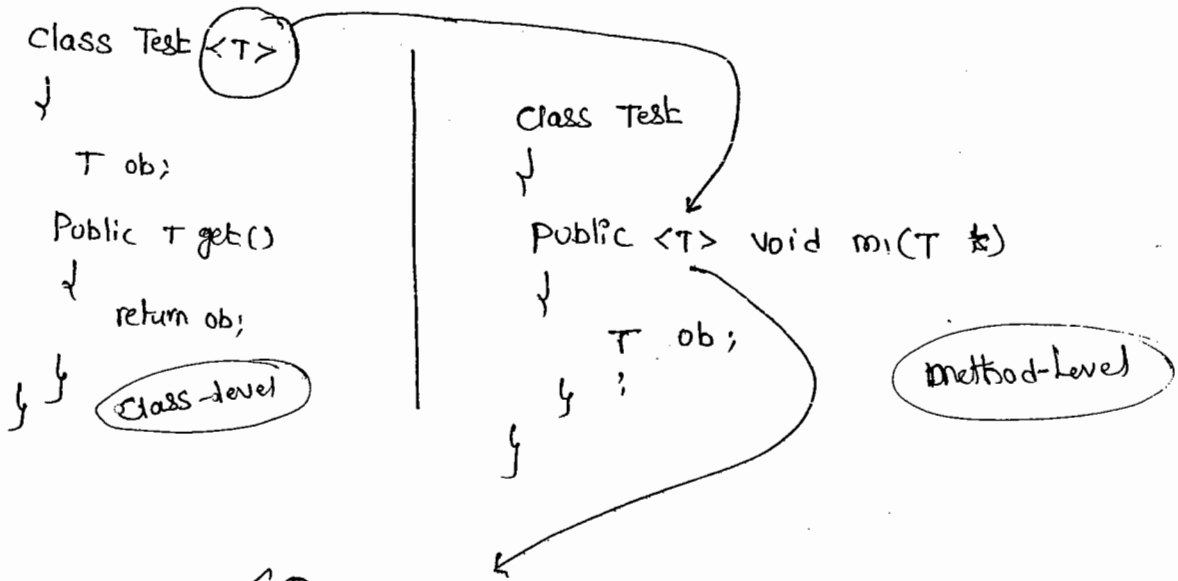
→ We can define the type parameter either at Class-Level or at method-Level.

Declaring type parameter at class level:-

```
Class Test <T>
{
    T ob;
    Public T get()
    {
        return ob;
    }
}
```

Declaring Type parameter at method-Level:-

→ we have to declare the type parameter just before returntype.



✓ ① <T extends Number>
✓ ② <T u Runnable>
✓ ③ <T u Number & Runnable>
✓ ④ <T extends Runnable & Comparable>
✗ ⑤ <T extends Number & Thread>
✗ ⑥ <T extends Runnable & Thread>

Communication with non-Generic Code :-

→ To provide Compatability with old version SUN people compromized the Concept of Generics in very few areas. the following is one such area.

```
eg:-    Class Test
        {
          P.S.V.m(String[] args)
          {
            AL<String> l = new AL<String>();
            l.add("A");
```

Ex:-

```
class Test
{
  p s.v.m(——)
  {
    AL<String> l = new AL<String>();
    l.add("A");
    // l.add(10); C.E
    m1(l);
    S.o.pln(l);    [A, 10, 10.5, true]
    // l.add(10); C.E
  }
  public static void m1(AL l)
  {
    l.add(10);    ✓
    l.add(10.5);  ✓
    l.add(true);
  }
}
```

Generic area (main method) | non-Generic area (m1 method)

Conclusions:-

Generics concepts is applicable only at Compiletime to provide type safety & to resolve type casting problems. At Runtime there is no such type of concept. Hence the following declarations are equal,

Ex:-

```
AL l = new AL();
AL l = new AL<String>();
AL l = new AL<Integer>();
```

✓ all are equal

ex:- ArrayList l = new ArrayList<String>();

```
l.add("A");
l.add(10);
l.add(true);
S.o.pln(l);  [A, 10, true]
```

→ The following two declarations are equal & there is no difference

both are equal

1) AL<String> l = new AL<String>();
2) AL<String> l = new AL();

09/04/2011

Multithreading

① Introduction

② The ways to define instantiate, and start a thread

③ Geting & Setting name of a thread

④ thread periorities

⑤ the methods to prevent thread execution

- yield()
- join()
- sleep()

⑥ Synchronization

⑦ Inter thread Communication

- wait()
- notify()
- notifyAll()

⑧ Deadlock

⑨ Daemon threads

Multitasking:-

→ Executing Several tasks Simultaneously is Called "multitasking".
There are 2 types of multitasking.

(1) process-based multitasking.

(2) thread-based multitasking.

Ex:- Students in ClassRoom.

(1) process-based multi-tasking:-

↦ Executing Several tasks Simultaneously, where each task is a Seperate independent process, is Called process based multitasking.

Ex:- While typing a Java program in editor we Can able to listen audio songs by MP3 player in the System. at the Same time we Can download a file from the net. all these tasks are Executing Simultaneously & independent of each other.
Hence, it is process-based multitasking.

→ process-based multitasking is best Suitable at "O.S Level."

(2) thread-based multitasking:-

→ Executing Several tasks Simultaneously where each task is a Seperate independent part of the Same program is Called "thread based multitasking" & each independent part is Called "thread".

↦ It is Best Suitable for "programatic Level".

→ Wheather it is process-based or thread-based the main objective of multitasking is to improve performance of the System by reducing Response-time.

→ the main important application areaas of multithreading are developing video games, multimedia Graphics, implementing animations,....

→ Java provides inbuilt Support for multithreading by introducing a Rich API (Thread, Runnable, ThreadGroup, ThreadLocal....). Being a programmer we have to know how to use this API and we are not responsible to define that API. Hence, developing multithreading programs is very easy when in Java when Compared with C++.

(2) The Ways to define, Instanciate & start a new thread :-

→ We can define a thread in the following 2 ways.

(i) By extanding Thread class.

(ii) By implementing Runnable Interface.

Defining a thread by extanding Thread class :-

defining a thread by Extending Thread class:-

Ex:-

```
Class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<=10; i++)
        {
            S.o.pln(" child thread");
        }
    }
}

Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    MyThread t = new MyThread(); // instantiation of thread
    t.start(); // Starting of a thread
    for(int i=0; i<=10; i++)
    {
      S.o.pln("main thread");
    }
  }
}
```

defining a thread

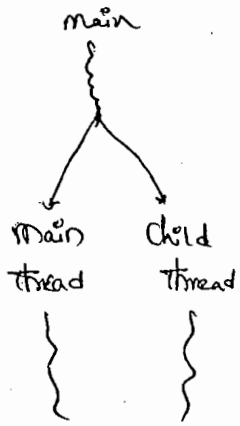
Executing child thread (MyThread)

main thread

child thread

Two threads

executing main thread



Case 1:-

Thread Schedulaer :-

→ when ever multiple threads are waiting to get chance for execution which thread will get chance first is decided by thread schedulaer whose behaviour is JVM vendor dependent. Hence we can't expect Exact Execution order & hence Exact o/p.

→ thread Schedulaer is the part of JVM. due to this unpredictable behaviour of thread schedulaer we can't Expect Exact o/p for the above program. the following are various possible o/p.

P-1

main thread
child thread

P-2

child thread
main thread

P-3

child thread
main thread
main thread
child thread
main thread

P-4

main thread
main 4
child 4
child 4
main thread

Note:-

→ when ever the situation comes to multithreading the guarantee in behaviour is very less. we can tell pasible o/p but not Exact o/p.

Case 2:-

difference b/w t.start() & t.run() :-

→ In the case of t.start() a new thread will be created & that thread is responsible to Execute run().

→ But in the Case of t.run() no new Thread will be Created & run method will be Executed Just like a normal method Call.

→ In the above program, if we are replacing t.start() with t.run() the following is the o/p.

o/p:-

Child thread
Child thread
10 times
main thread
10 times

} entire o/p produced by only main thread.

Case 3:-

Importance of Thread class Start() method!

→ To Start a thread, the required mandatory activities (like - registering thread with thread Scheduler) will be performed automatically by thread class Start() method. Because this facility, programmer is not responsible to perform this activity & he is just responsible to define Job of the thread. Hence thread class start() plays Very important role & without executing that method there is no chance of Starting a new thread.

Ex:-

```
class thread
{
    Start()
    {
        1. Register this thread with thread Scheduler & perform other
                                          initialization activities
        2. run()
    }
}
```

Case 4:-

* If we are not overriding run() method :-

→ if we are not overriding run() method then thread class run() will be executed which has Empty implementation & Hence we won't get any o/p.

Ex:-

```
Class Mythread extends Thread
{
}
class ThreadDemo
{
  P.S.V.m(String[] args)
  {
    Mythread t = new Mythread();
    t.start();
  }
}
```

O/P:- no o/p printing

Note:

* It is highly recommended to override run() to define our Job.

Case 5:-

Overloading of run() :-

→ Overloading of the run() is possible, but thread class start() will always call no argument run() only. but the other run() we have to call Explicitly Just like a normal method call.

Eg:-

```
Class MyThread extends Thread
{
    Public void run()
    {
     S.o.pln("run()");
    }
    public void run(int i)
    {
     S.o.pln("run(int i)");
    }
}

Class ThreadDemo
{
 p.s.v.m(String[] args)
 {
  MyThread t = new MyThread();
   t.start();
 }
}
```

(overloading)

o/p:- run()

Case 6:-

Overriding of start():-

→ If we override start() then start() will be executed just like a normal method call & no new thread will be created.

Ex:-

```
Class MyThread extends Thread
{
  Public void start()
  {
```

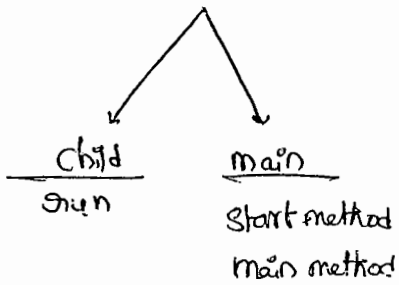
```
        S.o.pln("Start method");
        }
        Public void run()
        {
          S.o.pln("run");
        }
      }
      Class ThreadDemo
      {
        p.s.v.m(String[] args)
        {
          MyThread t = new MyThread();
          t.start();
        }
      }
```

O/P:- Start method.

Case(I):-

```
      class MyThread extends Thread
      {
        public void start()
        {
          Super.start();
          S.o.pln("start method");
        }
        public void run()
        {
          S.o.pln("run");
        }
      }
```

```
Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    MyThread t = new MyThread();
    t.start();
    S.o.println("main method");
  }
}
```

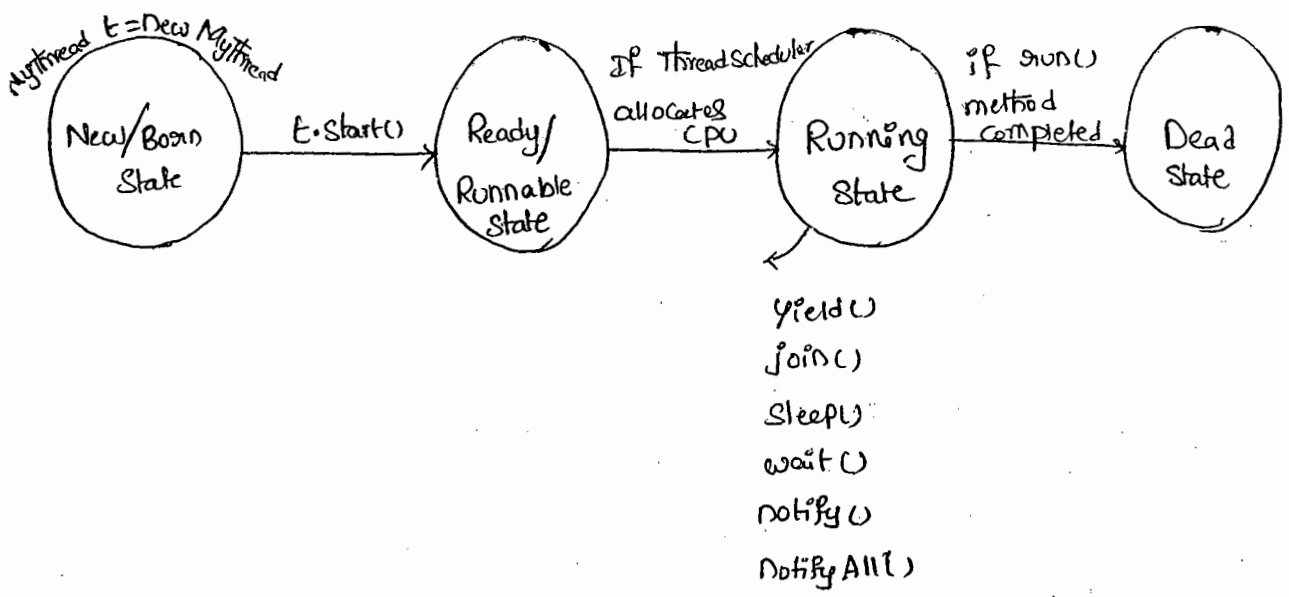


o/p:-

P-1 ✓	P-2 ✓	P-3 ✓	P-4 ✗
Start method	run	Start method	main method
run	Start method	Main method	Start method
Main method	main method	run	run

Case-8:-

* Life Cycle of a thread :-



→ Once we Created a thread Object then it is Said to be in New State or born State.

→ If we Call Start() method then the thread will be entered into Ready (or) Runnable State.

→ If ThreadScheduler allocates cpu, then the thread will entered into Running State.

→ If run() method Completes then the thread will entered into DeadState

*Case 9:-

→ After Starting a thread we are not allowed to restart the Same thread once again otherwise we will get IllegalRuntimeException Saying "IllegalThreadStateException".

eg:

```
Thread t = new Thread()
t.start();
:
:
t.start(); X   R.E:- IllegalThreadStateException.(ITSE)
```

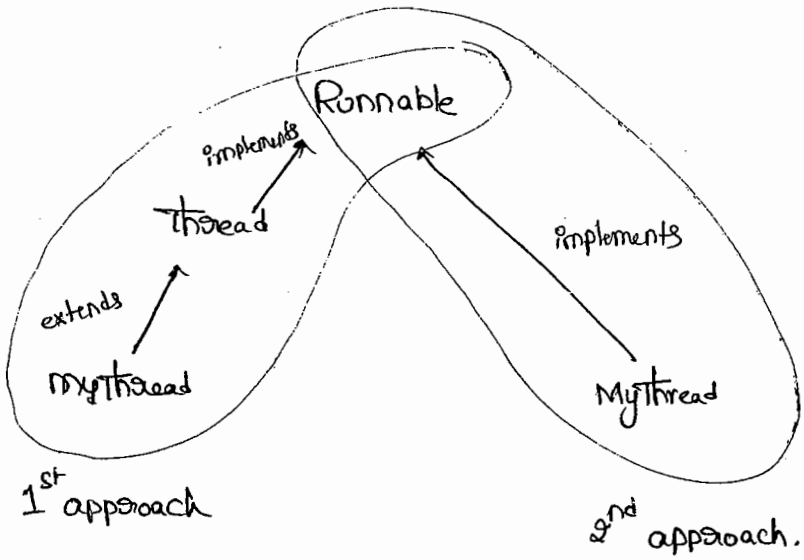
→ with in the run() if we Call Super.start() we will get the Same run-time Exception.

Note:-

→ JVM never recommended to override start(), but it is highly recommended to override run().

(2) Defining a Thread by implementing "Runnable Interface":-

* We can define a Thread even by implementing Runnable Interface also.
* Runnable Interface present in Java.lang package & Contains only one method run() method.




```
Runnable
{
  run()
  {
  }
}

class Thread
{
  start()
  {
    run()
  }
}
```

Ex:-

```
Class MyRunnable implements Runnable
{
   public void run()
   {
      for(int i=0 ; i<10 ; i++)
      {
        S.o.pln("child Thread");
      }
   }
}
```

defining a thread is Great

Job of thread

```
Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    MyRunnable r = new MyRunnable();
    Thread t = new Thread(r);
                    └──→ target Runnable
    t.start();
  ──→
    for(int i=0; i<=10; i++)
    {
      S.o.pln("main thread");
    }
}}
```

→ We can't get Exact o/p & we will get mixing o/p

Case Study :-

```
MyRunnable r = new MyRunnable();
Thread t1 = new Thread();
Thread t2 = new Thread(r);
```

Case(1) :-

(i) t_1.start() :-

→ A new thread will be created which is responsible for Execution of Thread class run().

Case(2) :- t_1.run() :-

→ No new thread will be Created & Thread class run() will be Executed Just Like a normal method call.

Case 3:- t2.start():-

→ New thread will be Created which is responsible for the Execution of MyRunnable run() method.

Case 4:- t2.run():-

→ No new thread will be Created & MyRunnable run() will be Executed just like a normal method Call.

Case 5:- r.start():-

→ We will get Compile-time Error Saying Start() is not available in MyRunnable class

C.E:- Cannot find Symbol

Symbol : method Start()

location : class MyRunnable

Case 6:- r.run():-

→ No new thread will be Created & myRunnable run() will be Executed Just like a normal method Call.

Q) In which of the above cases a new thread will be Created

A) t1.start() & t2.start()

Q) In which of the above Cases MyRunnable class run() will be executed Just like a normal method &

t2.run() & r.run()

Best Approach to define a Thread :-

→ Among the two ways of defining a thread implements Runnable mechanism is recommended to use.

→ In the first approach, ~~thread~~ our class always extending Thread class & hence there is no chance of Extending any other class. But in the second approach we can Extend some other class also while implementing Runnable interface. Hence 2nd approach is Recommended to use.

Thread class Constructors :-

(1) Thread t = new Thread();

(2) Thread t = new Thread(Runnable r);

(3) Thread t = new Thread(String name);

(4) Thread t = new Thread(Runnable r, String name);

(5) Thread t = new Thread(ThreadGroup g, String name);

(6) Thread t = new Thread(ThreadGroup g, Runnable r);

(7) Thread t = new Thread(ThreadGroup g, Runnable r, String name);

(8) Thread t = new Thread(ThreadGroup g, Runnable r, String name, long stackSize);

Durgas approach to define a Thread (not recommended to use):

ex:-

```
Class MyThread extends Thread
{
  public void run()
  {
    S.o.pln("run method");
  }
}
Class Test
{
  P.S.V.m(String[] args)
  {
    MyThread t = new MyThread();
    Thread t1 = new Thread(t);
      t1.start();
    }S.o.pln("main");
  }
}
```

o/p:-

run	main
main	run

3) Getting & Setting name of a thread :-

→ Every thread in Java has some name. It may be provided by the programmer or default name generated by JVM.

→ We can get & set name of a thread by using the following methods of Thread class.

(i) public final String getName();

(ii) public final void setName(String name);

Ex:-

```
class Test
{
  p.s.v.m(String[] args)
  {
    S.o.println(Thread.currentThread().getName()); // main
    Thread.currentThread().setName("pavabas");
    S.o.println(Thread.currentThread().getName()); // pavabas.
  }
}
```

Note:-

→ We can get current executing thread Refexere by using the following method of thread class.

public static thread currentThread();

4) Thread priorities:-

→ Every thread in Java has some priority but the range of thread priorities is "1 to 10". (1 is least & 10 is highest).

→ Thread class defines the following constants to define some standard priorities.

1) Thread.MIN_PRIORITY ⟶ 1
2) Thread.NORM_PRIORITY ⟶ 5
3) Thread.MAX_PRIORITY ⟶ 10
4) Thread.LOW_PRIORITY
5) Thread.HIGH_PRIORITY

→ Thread Schedular will use these priorities while allocating Cpu

→ The thread which is having highest priority will get chance first.

→ If two threads having same priority then we can't expect exact execution order, it depends on thread schedular.

default priority:-

→ The default priority only for the main thread is 5. But for all the remaining threads it will be inheriting from the parent. i.e whatever the priority parent has the same priority will be inheriting to the child.

* Thread class defines the following 2 methods to get & set priority of a thread,

① public final int getPriority();

② public final void setPriority(int p);

→ the allowed values are 1 to 10, otherwise we will get IllegalArgumentException.

Ex:-

t.setPriority(5); ✓

t.setPriority(10); ✓

✗ t.setPriority(100); ✗ R.E:- IAE (IllegalArgumentException).

Ex:-

```
class MyThread extends Thread
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      S.o.pln("Child thread");
    }
  }
}

class ThreadPriorityDemo
{
  p.s.v.m(String[] args)
  {
    MyThread t = new MyThread();
    // t.setPriority(10);  ——→ ①
    t.start();
    for(int i=0; i<=10; i++)
    {
      S.o.pln("main method");
    }
  }
}
```

→ If we are Commenting line① then Both main & child threads having the Same priority (5) & Hence we Can't Expect Exact Execution order and Exact o/p.

→ If we aren't Commenting line① then main thread has the priority 5 & child thread has the priority 10 & Hence child thread will be Executed first & then main thread. In this Case the o/p is

```
child thread
    ≡ 10 times
main thread
    ≡ 10 times
```

The methods to prevent Thread Execution :-

→ we can prevent a thread from Execution by using the following methods.

(i) yield()
(ii) join()
(iii) sleep()

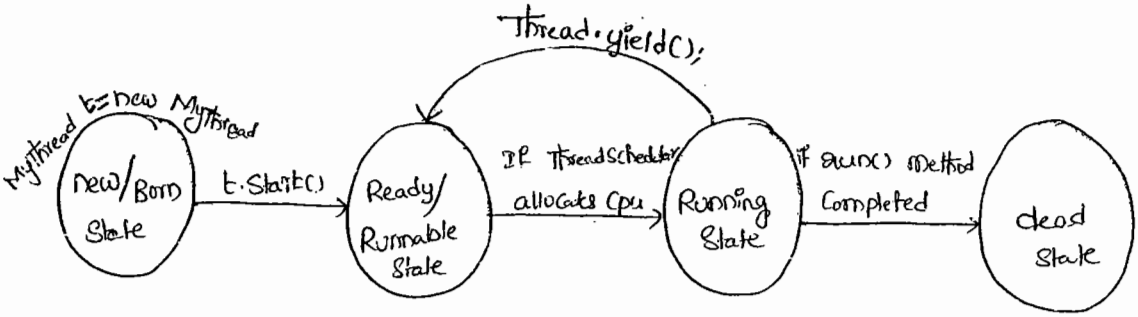
(i) yield() :-

→ yield() method Causes, to pause Current Executing thread for giving the chance to remaining waiting threads of Same priority.

→ If there are no waiting threads or all waiting threads have low priority then the Same thread will Continue it's execution once again.

→ Signature of yield() method

```
public static native void yield()
```



→ The thread which is yielded, when it will get chance once again for Execution is decided by ThreadScheduler. & we can't Expect Exactly.

Ex:-

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<10; i++)
        {
            Thread.yield();  ----------> ①
            S.o.println("child thread");
        }
    }
}
class ThreadYieldDemo
{
    p.s.v.m(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        for(int i=0; i<10; i++)
        {
            S.o.println("main thread");
        }
    }
}
```

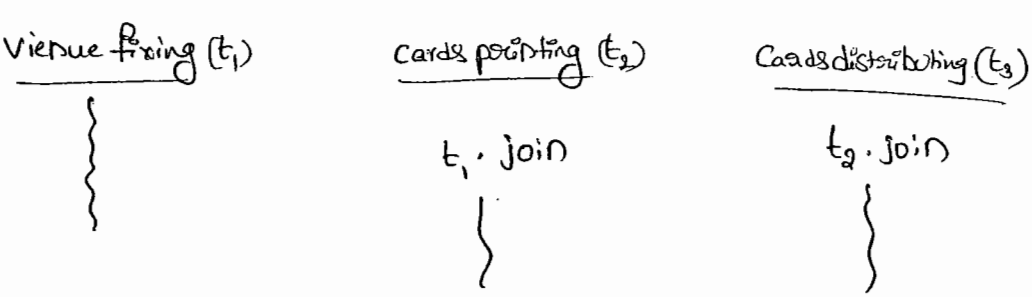
→ If we are Commenting Line① the both threads will be Executed Simultaneously & we Can't Expect Exact Execution Order.

→ If we are not Commenting Line① then the chance of Completing main thread first is high because child thread always calls yield().

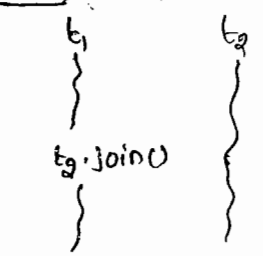
ii) join() :-

→ If a thread wants to wait until Completing Some other thread then we should go for join() method.

Ex:-(1)



Ex(2):-



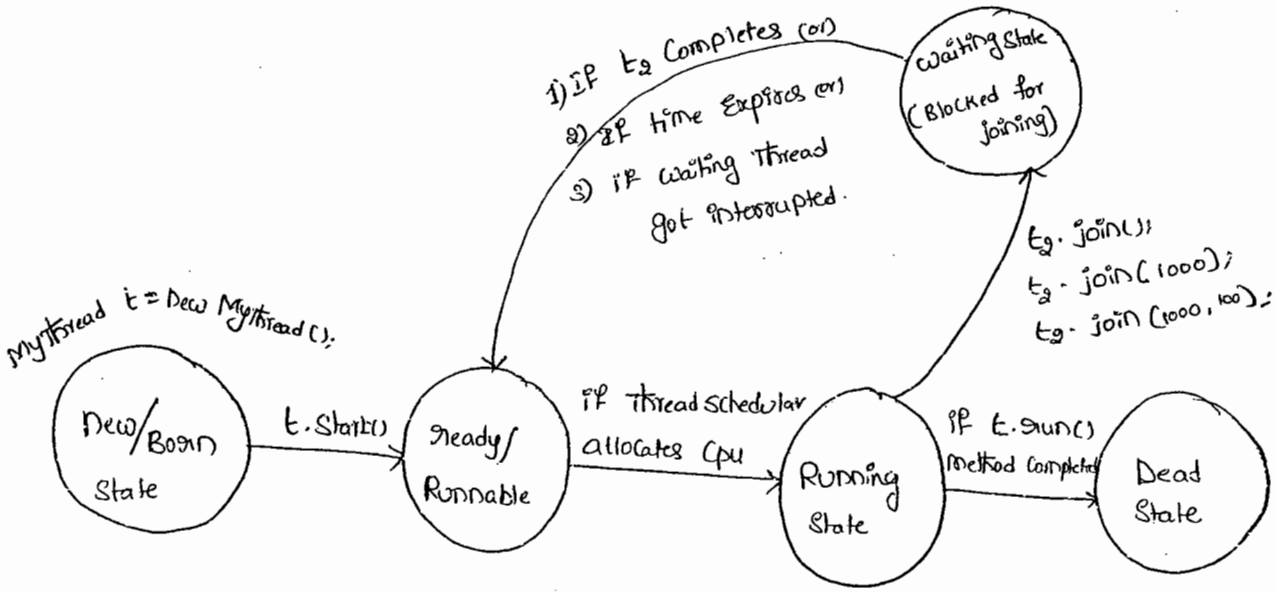
→ If thread t_1 Executes t_2.join() then t_1 thread will entered into waiting state until t_2 Completes. Once t_2 Completes then t_1 will Continue its execution.

(i) public final void join() throws InterruptedException

(ii) public final void join(long ms) throws InterruptedException

(iii) public final void join(long ms, int ns) throws InterruptedException.

→ join() method is Overloaded and Every join() throws InterruptedException. Hence, when ever we are using join() Complusary we should handle InterruptedException, either by try-catch or by throws other wise we will get Compiletime Error.




```
Class MyThread extends Thread
{
  Public void run()
  {
    for(int i=0 ; i<10 ; i++)
    {
      S.o.pln("Sitha Thread");
      try
      {
        Thread.Sleep(2000);
      }
      Catch(IE e)
      {
      }
    }
  }
Class ThreadJoinDemo
{
  p.s.v.m(String[] args) throws InterruptedException
  {
    MyThread t1 = new myThread();
    t1.Start();
    t1.join(); ——→ ①
```

```
for(int i=0 ; i<10 ; i++)
{
  S.o.pln("Rama thread");
}
}}}
```

→ If we are Commenting Line① then both threads will be Executed Simultaneously and we Can't Expect Exact Execution Order. And Hence we can't Expect Exact o/p.

→ If we are not Commenting Line① then main thread will wait until Completing child thread. Hence in this case the o/p is Expected.

o/p :- Sitathread 10 times
≡
Ramathread 10 times
≡

(iii) Sleep() :-

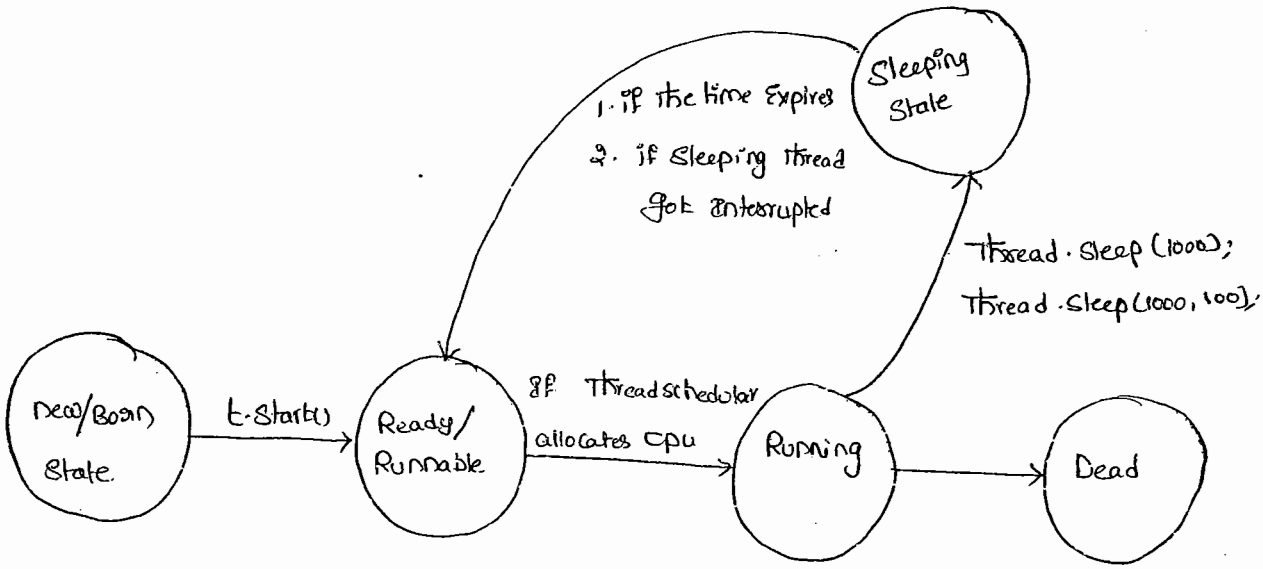
→ If a thread don't wont to perform any operation for a particular amount of time (Just pausing) then we should go for Sleep()

1) public static void sleep(long ms) throws InterruptedException

2) public static void sleep(long ms, int ns) throws InterruptedException

→ when ever we are using Sleep() method Compolsary we should handle InterruptedException other wise we will get Compiletime Error.

Static :- because sleep method calls thread.sleep() means classname
t1.start(); t1 is object so it is instance (a) non static



Ex:-
```
Class Test
{
 p.s.v.m(String[] args) throws InterruptedException
 {
   S.o.pln("Durga");
   Thread.sleep(5000);
   S.o.pln("Software");
   Thread.sleep(5000);
   S.o.pln("Solutions");
 }}
```

Interruption of a Thread :-

* A Thread Can Interrupt another Sleeping or waiting Thread.
* for this Thread class defines interrupte() method.

```
public void interrupte()
```

Ex:-

```
class MyThread extends Thread
{
  Public void run()
  {
   try
    {
    for (int i=0 ; i<100; i++)
     {
      S.o.pln(" Lazy Thread");
     Thread.sleep(5000);
     }
   }
  Catch(IE e)
  {
   S.o.pln(" I got Interrupted");
  }
}
Class InterruptDemo
{
  P.S.v.m(String[] args)
  {
    MyThread t = new MyThread();
      t.start();
  →   t.interrupt(); ——→ ①
      S.o.pln("end of main");
  }}
```

→ If we are Commenting line ① then main thread won't Interrupte Child thread Hence both threads will be executed until Completion

→ If we are not Commenting line① then main thread interrupts the Child thread ~~here child thread went Sleep~~ raises Interrupted Exception.

→ In this case the o/p is

o/p:- I am Lazy thread

I got Interrupted

End of main

**Note:-

* → We ~~can't~~ may'nt See the impact of interrupt call immediately.

* → when ever we are Calling interrupt() method, if the target thread is not in Sleeping or waiting state then there is no impact immediately Interrupt Call will wait until target thread entered into Sleeping or waiting State. Once target thread entered into Sleeping (or) waiting state the interrupt call will impact the target thread.

* Comparision table for yield(), join(), Sleep() :-

Property	yield()	join()	Sleep()
1) Purpose ?	to pause Current executing thread to give the chance for the remaining threads of same priority.	if a thread want to wait until Completing Some other thread then we should go for join	If a thread don't want to perform any operation for a particular amount of time (pausing) go for Sleep()
2) Static	yes	No	yes
3) Is it over-loaded	No	yes	yes
4) Is it final	No	yes	No
5) Is it throws InterruptedException	No	yes	yes
6) Is it native Method	yes	No	Sleep(long ms) → native Sleep(long ms, int ns) → non-native

Synchronization :-

→ Synchronized is the modifier applicable only for methods & blocks & we can't apply for classes & variables.

→ If a method or block declared as Synchronized then at a time only one thread is allowed to execute that method or block on the given object.

→ The main advantage of Synchronized key-word is we can resolve data inconsistency problem.

→ The main limitation of Synchronized keyword is it increases waiting time of the threads & effects performance of the System. Hence if there is no specific requirement it's never recommended to use Synchronized key-word.

→ Every object in Java has a unique lock Synchronization Concept internally implemented by using this Lock Concept. when ever we are using Synchronization then only Lock Concept will come into the picture.

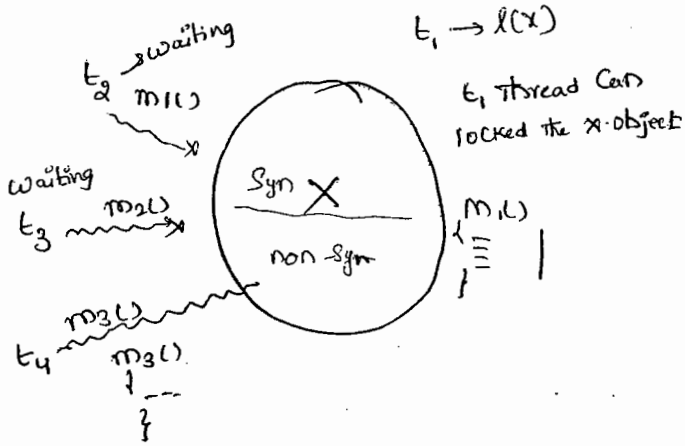
→ If a thread wants to execute any Synchronized method on the given object, first it has to get the lock of that object. Once a thread get a lock then it allowed to Execute any Synchronized method on that object.

→ Once Synchronized method Completes then automatically the lock will be released.

↳ while a thread Executing any Synchronized method on the given object the remaining threads are not allowed to Execute any Synchronized method on the given Object ~~Symmit~~ Simultaneously. But remaining threads are allowed to Execute any non-Synchronized methods Simultaneously (lock concept is implemented based on Object but not based on method).

Ex:-

```
Class X
{
    Sync m1()
    {
    }
    Sync m2()
    {
    }
    m3()
    {
    }
}
```

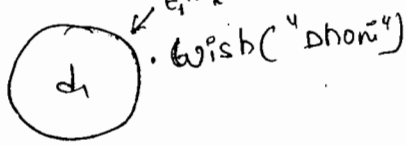


Lock:-
It is an Object associated with for every Object

Ex:-

```
Class Display
{
public  Synchronized  void wish(String name)
{
 for(int i=0; i<10; i++)
 {
   S.o.print(" Good morning: ");
   try
   {
     Thread.sleep(3000);
   }
   Catche(IE e) { }
```

```
        S.O.ptn(name);
    }
}
Class MyThread extends Thread
{
    Display d;
    String name;
    MyThread(Display d, String name)
    {
        this.d = d;
        this.name = name;
    }
    public void run()
    {
        d.wish(name);
    }
}
Class SynchronizedDemo
{
    p.s.v.m(String[] args)
    {
        Display d1 = new Display();
        MyThread t1 = new MyThread(d1, "Dhoni");
        MyThread t2 = new MyThread(d2, "Yuvaraj");
        t1.start();
        t2.start();
    }
}
```



→ If we are not declaring wish() method as Synchronized then both Threads will be Executed Simultaneously & we Can't Expect Exact o/p we will get irregular o/p.

o/p:-

```
Goodmorning : Goodmorning : Dhoni
Goodmorning : yuvaraj
    "       : Dhoni
    "
```

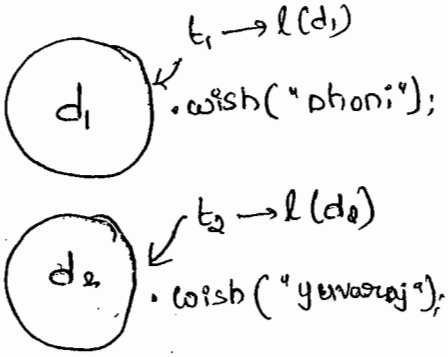
→ If we declare wish() method as Synchronized then Threads will be Executed one by one So that we will get Regular o/p.

o/p:-

```
Goodmorning : Dhoni
    :
  10times
Goodmorning : yuvaraj
    :
  10times
```

Case Study:-

```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "yuvaraj");
t1.start();
t2.start();
```



→ Eventhough wish() method is Synchronized we will get irregular O/P in this case. Because, the Threads are operating on different Objects.

Reason:-

→ Whenever multiple Threads are operating on same Object then only Synchronization play the role. If multiple Threads are operating on multiple Objects then there is no impact of Synchronization.

Classlevel Lock:-

→ Every class in Java has a unique lock,

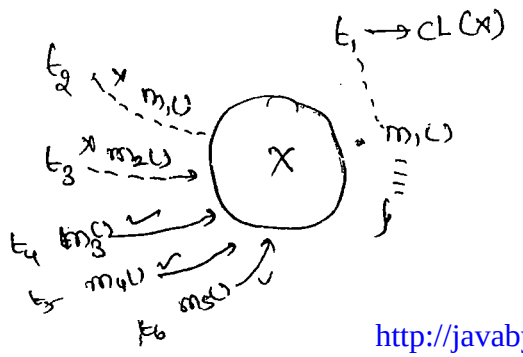
→ If a thread wants to Execute a Static Synchronized method then it required classlevel lock.

→ While a thread executing a Static Synchronized method then the remaining threads are not allowed to execute any Static Synchronized method of that class simultaneously but remaining threads are allowed to execute the following methods simultaneously.

✓ 1. Normal Static methods.
✓ 2. Normal instance methods.
✓ 3. Synchronized instance methods.

Ex:-

```
Class X
{
Static Syn m1()
Static Syn m2()
    Syn m3()
    Static m4()
    m5()
}
```



Note :-

→ There is no link between object Level lock & class Level Lock both are independent of Each other.

→ ClassLevel lock is different & ObjectLevel lock is different.

Synchronized Block :-

→ If very few lines of code requires Synchronization then it is never recommended to declare entire method as Synchronized, we have to declare those few lines of code inside Synchronized block

→ the main Advantage of Synchronized Block over Synchronized method is, it reduces the waiting time of the threads & improves performance of the System.

Ex(1) :-

→ We can declare Synchronized block to get Current Object lock as follows.

```
Synchronized (this)
{
.....
}
```

→ If thread got lock of Current object then only it is allowed to Execute this block.

Ex(2) :-

→ To get lock of a particular object b we can declare Synchronized block as follows.

```
Synchronized(b)
{
  ==========
}
```

→ if thread get lock of 'b' then only it is allowed to Execute that blocks.

Ex(3):-

→ To Get class level lock we can declare Synchronized block as follows.

```
Synchronized(classname.class)
{
  ---
  ===
}
```

→ if thread got Classlevel lock of classname (ex Display) class then only it is allowed to Execute that block.

Ex(4):-

Synchronized block concept is applicable only for Objects & classes but not for. premitives otherwise we will get Compiletime Error.

```
int x=10;
Synchronized(x)
{
  ---
  ===
}
```

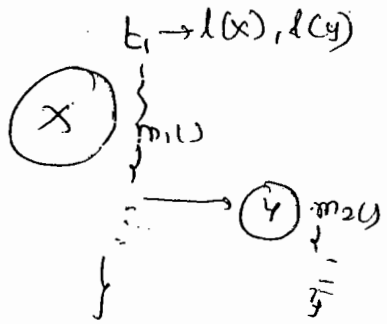
C.E:- Unexpected type
found : int
Required : Reference

→ Every object in java has a unique lock, But a thread can acquire more than one lock at a time (ofcourse from diff. objects)

Eg:

```
Class X
{
Syn m1()
{
==
y y = new Y();
y.m2();
==
}
}
```

```
Class Y
{
Syn m2()
{
==
}
}
```



FAQ!:-

① Explain about Synchronized Keyword & What are various Advantages & disAdvantages?

② what is Object lock & when it is required?

③ While a thread Executing an instance Synchronized method on the given Object then is it possible to Execute any other Synchronized method Simultaneously by other threads? Ans:- Not possible

④ What is ClassLevel Lock & when it is required.

⑤ What is the diff. blw Object lock & class Level locks

⑥ what is the advantage of Synchronized block over Synchronized method

⑦ How to declare Synchronized block to get class level locks?

⑧ What is Synchronized Statement? (Interview people Created terminology)

⇒ The Statements present in Synchronized method & Synchronized blocks are called as Synchronized Statement.

30/04/11

Inter Thread Communication :-

↳ Two threads will Communicate with each other by using wait(), notify(), notifyAll() methods. The thread which requires updation it has to call wait() method. The thread which is responsible to update it has to call notify() method.

→ wait(), notify(), notifyAll() methods are available in Object class but not in thread class. because Threads are required to call these method on any shared object.

* If a thread wants to call wait(), notify(), & notifyAll() methods Compulsory the thread should be owner of the object. i.e, the Thread has to get lock of that object. i.e, the thread should be in the Synchronized area.

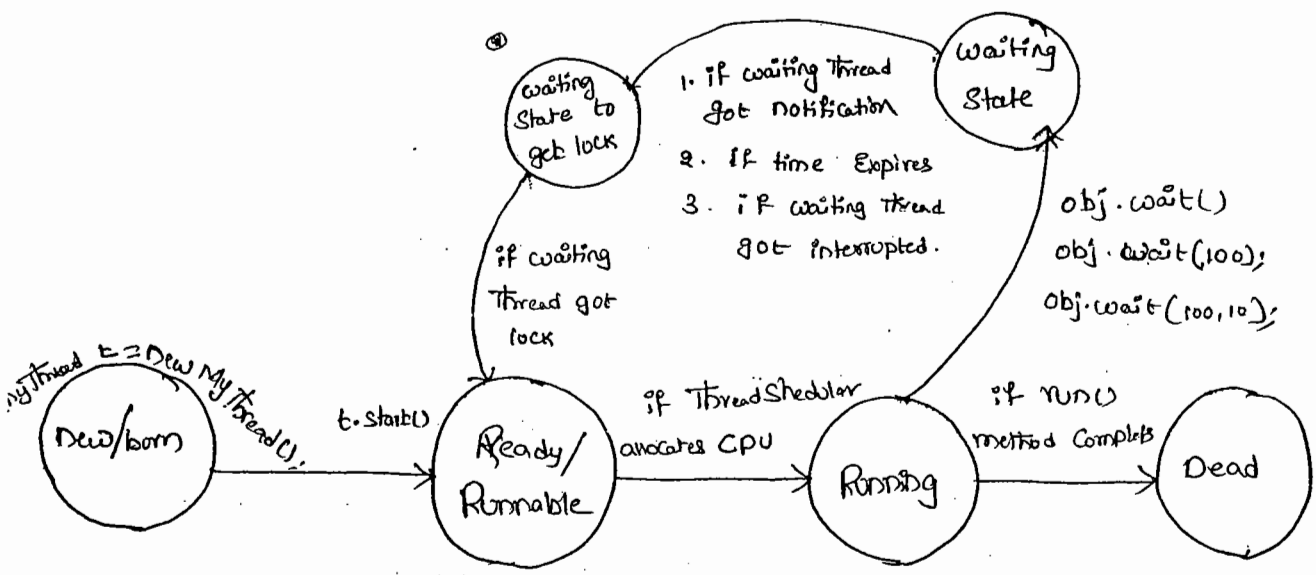
→ Hence, we can call wait(), notify(), notifyAll() methods only from Synchronized area otherwise we will get RuntimeException saying "IllegalMonitorStateException".

→ If a thread calls wait() method it releases the lock immediately and entered into waiting state. A thread releases the locks of only Current Object but not all locks. After calling notify() and notifyAll() methods Thread releases the lock but may not immediately. Except these wait(), notify(), notifyAll() there is no other case where thread releases the lock.

javap java.lang.Object

method	is thread releases lock ?
yield()	No
join()	No
sleep()	No
wait()	yes
notify()	yes
notifyAll()	yes

1) public final void wait() throws IE
2) public final native void wait(long ms) throws IE
3) public final void wait(long ms, int ns) throws IE
4) public final native void notify()
5) public final native void notifyAll()



Ex:-

```
Class ThreadA
{
 P.S.V.m(String[] args)throws InterruptedException
 {
   ThreadB b = new ThreadB();
   b.start();
   Synchronized (b)  ------> Thread.sleep(1000);
   {
 ① S.o.pln("main Thread trying to Call wait()");
   b.wait();   // b.wait(1000);
 ④ S.o.pln("main thread got notification");
 ⑤ S.o.pln(b.total);
    }
   }
Class ThreadB extends ThreadA
{
  int total=0;
  Public void run()
  {
   Synchronized (this)
  }
 ② S.o.pln("child Thread strats notification");
   for(int i=1 ; i<=100 ; i++)
   {
     total = total + i;
   }
 ③ S.o.pln("child thread trying to given notification");
   this.notify();
}}}
```

O/P:- main Thread Calling wait method

Child Thread starts Calculation

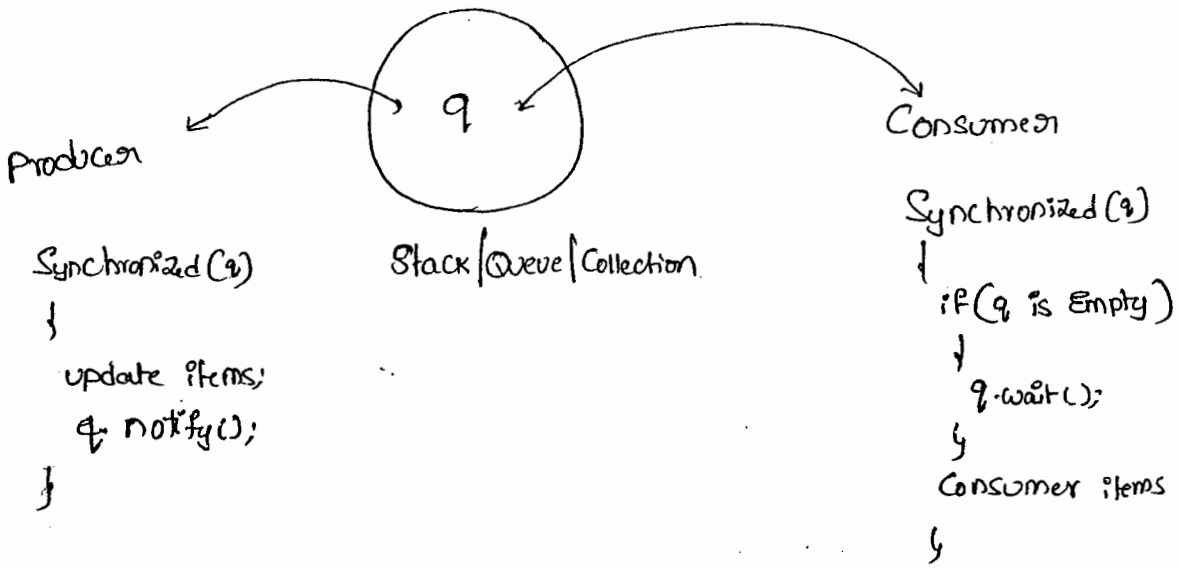
Child giving notification

Main Thread got notification

5050

Ctrl+C

Producer-Consumer problem:-



↳ Consumer has to Consume items from the Queue

↳ if Queue is Empty, he has to call wait() method.

↳ producer has to produce items into the Queue.

↳ After producing the items, he has to call notify() method so that all waiting Consumers will get notification.

notify() vs notifyAll():

→ We can use notify() to notify only one waiting thread. But which waiting thread will be notified we can't expect exactly. All remaining threads have to wait for further notifications.

→ But in the case of notifyAll() all waiting threads will be notifying but the threads will be executed one by one.

*Note:-

→ On which object we are calling wait(), notify() & notifyAll(), we have to get the lock of that object.

```
Stack s1 = new Stack();
Stack s2 = new Stack();
```

```
synchronized(s1)
{
  ==
  ==
  s2.wait();
}
```

R.E: IllegalMonitorStateException

```
synchronized(s1)
{
  ==
  s1.wait();
}
```

DeadLock :-

→ If two threads are waiting for each other for ever. Such type of situation is called "DeadLock."

→ There are no resolution techniques for deadLock but several prevension techniques are possible.

ex:-

```
class A
{
 public synchronized void foo(B b)
 {
  S.o.pln("thread1 starts execution foo");
  try
  {
   Thread.sleep(1000);
  }
  Catch(IE e)
  {
  }
  S.o.pln("thread1 taying to Catch b's last()");
  b.last();
 }
 public Synchronized void last()
 {
  S.o.pln(" Inside A this is last()");
  }
}
```

```
Class B
{
 Public Synchronized void bar(A a)
 {
  S.o.pln("Thread2 starts bar");
  try {
       Thread.sleep(5000);
      }
  Catch(IE e) { }
 S.o.pln("Thread 2 trying to call a's last");
  a.last();
 }
 Public synchronized void last()
 {
  S.o.pln("Inside B this is last");
 }
}

Class DeadLock extends Thread
{
   A a = new A();
   B b = new B();

   DeadLock()
   {
     this.Start();
     a.foo(b);   // executed by main thread
   }
   Public void run()
   {
     b.bar(a);  // executed by child thread
   }
   P.S.V.m(——).
}}  new DeadLock()}
```

o/p :-

Thread1 Starts execution of foo method

Thread2 Starts execution of bar method

Thread#1 trying to Call b's last()

Thread2 trying to Call a's last()

:::

→ Synchronized keyword is the only one reason for deadlock hence while using Synchronized keyword we have to take very much Care.

* DeadLock Vs Starvation :-

→ In the Case of Deadlock waiting never ends.

→ A Long waiting of a thread which ends at certain point of time is Called "Starvation."

Ex :-

least priority Thread has to wait until Completing all the Threads but this long waiting should Compulsory ends at certain point of time.

→ Hence, A long waiting which never Ends is Called "DeadLock" where as a Long waiting which Ends at certain point of time is Called "Starvation"

Daemon Threads :-

→ The threads which are executing in the background are called "Daemon threads". Ex:- Garbage Collector

→ The main objective of Daemon threads is to provide support for non-Daemon threads.

→ We can check whether the thread is Daemon or not by using "isDaemon() method."

```
public final boolean isDaemon()
```

→ We can change Daemon nature of a thread by using setDaemon() method

```
public final void setDaemon(boolean b)
```

→ We can change Daemon nature of a thread before starting only. If we are trying to change after starting a thread we will get RuntimeException Saying "IllegalThreadStateException".

→ main thread is always non-Daemon & its not possible to change it's Daemon nature.

Default nature :-

→ By default main thread is always non-daemon but for all the remaining threads Daemon nature will be inheriting from parent to child. i.e, if the parent is Daemon, child is also Daemon & if the parent is non-Daemon then child is also non-Daemon.

→ whenever the last non-Daemon thread terminates all the Daemon threads will be terminated automatically.

Ex:-

```
Class MyThread extends Thread
{
    Public void run()
    {
        for (int i=0 ; i<10 ; i++)
        {
            S.o.pln("lazy thread");
            try {
                Thread.sleep(2000);
            }
            Catch (InterruptedException e) { }
        }
    }
Catch
Class Test
{
    P.S.V.m(String[] args)
    {
        MyThread t = new MyThread();
        t.setDaemon(true);  ——— ①
        t.start();
        S.o.pln("end of main");
    }
}
```

→ If we are Commenting Line① Then both main & child threads are non-Daemon & hence both will be executed until their Completion.

→ If we are not Commenting Line① then main thread is non-Daemon & child Thread is Daemon. Hence when ever main thread terminates automatically child thread will be terminated.

How to kill a Thread :-

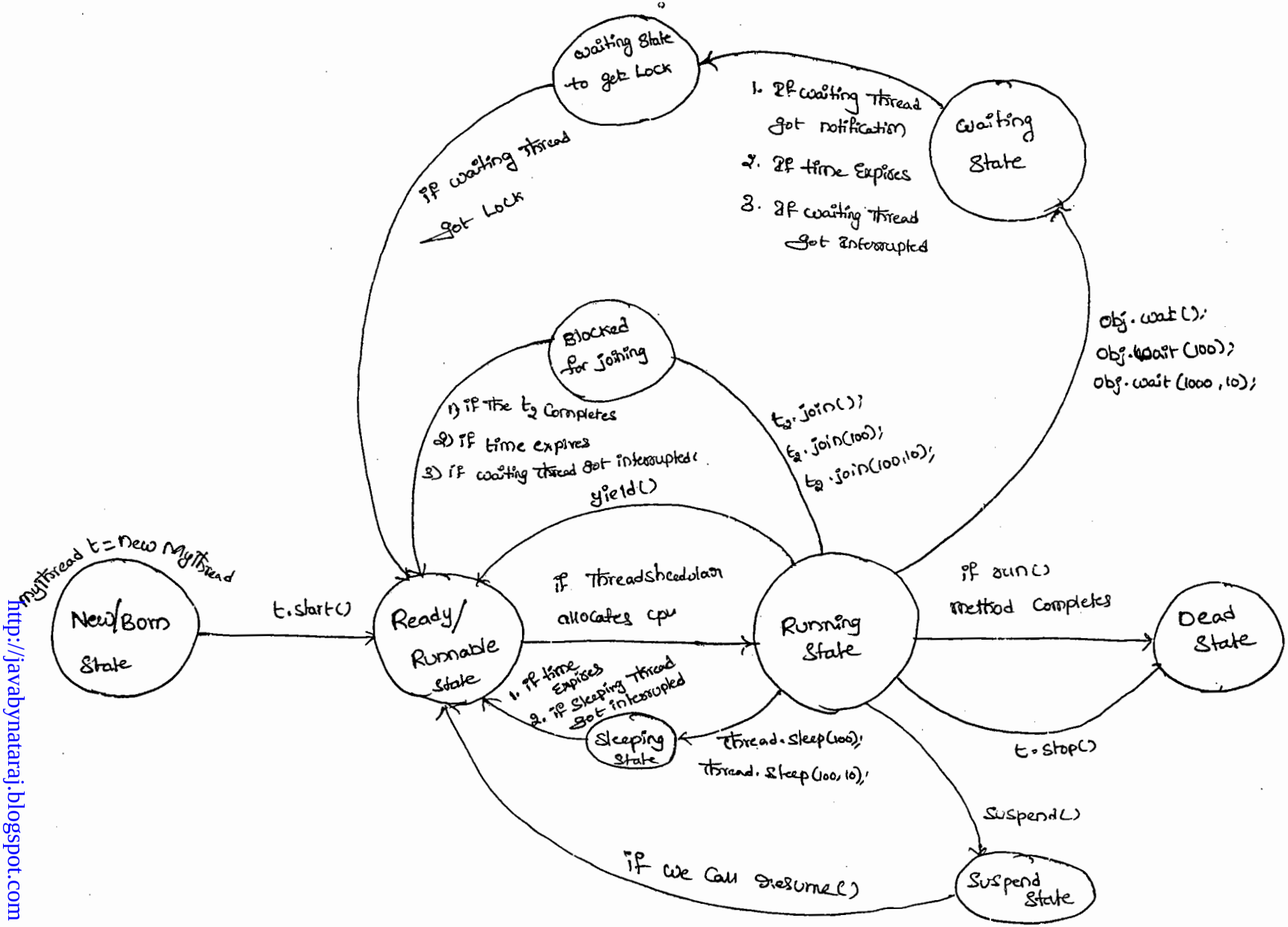
→ A Thread Can Stop or Kill another Thread by using Stop() method then automatically running Thread will entered into Dead State. It is a deprecated method & Hence not recommended to use.

```
public void Stop();
```

Suspending & Resuming a thread :→

→ A Thread Can Suspend another thread by using Suspend() method.
→ A Thread Can resume a Suspended Thread by using resume() method.
→ Bot these methods are deprecated methods & Hence not recommended to use.

a) What is a Green Thread?
b) What is ThreadLocal?



Case 4 :-

```
Class Test
{
 Public void m1(int i, float f)
 {
  S.o.pln(" int-float version");
 }
 Public void m1(float f, int i)
 {
  S.o.pln(" float-int version");
 }
 P.S.v.m(___)
 {
  Test t1 = new Test();
  t1.m1(10,10.5F);
  t1.m1(10.5f,10);
  t1.m1(10,10);   X C.E!- reference to m1() is ambiguous
  t1.m1(10.5F,10.5F); X C.E!-
 }
}
```

Can not find Symbol

Symbol: method m1(float, float)

location: Class Test.

Case 5 :-

```
Class Animal
{
}
Class Monkey extends Animal
{
}
Class Test
{
  Public void m1(Animal A)
  {
    S.o.pln("Animal Version");
  }
  public void m1(Monkey m)
  {
    S.o.pln("monkey version");
  }
  P.S.V.m(——)
  {
    Test t = new Test();
    Animal a = new Animal();
      t.m1(a); // Animal version
    Monkey m = new Monkey();
      t.m1(m) // monkey-
    Animal a1 = new monkey();
      t.m1(a1); // Animal
  }
}
```

→ In overloading method resolution always takes place by Compiler based on reference type and Runtime Object never play any role in overloading.

Over riding :→

03/05/11

→ "What ever the parent has by default available to the child. if the child not satisfied with its parent class implementation then child is allowed to redefine its implementation in its own way." this process is called "overriding"

→ The parent class method which is overridden is called overridden method & the child class method which is overriding is called overriding method.

Ex:-

```
Class P
{
    public void property()
    {
      S.o.pln("Cash + Gold + Land");
    }
    Public void marry()
    {
      S.o.pln("Subba Laxmi");
    }
}

Class C extends P
{
    Public void marry()
    {
      S.o.pln("Kajal | 3sha | 4tara | 4me");
    }
}
```

overridden method

overriding

overriding method

Ex2:-


```
class P
{
    public void m1()
    {
        S.o.pln("parent");
    }
}
Class C extends P
{
    public void m1()
    {
        S.o.pln("child");
    }
}
Class Test
{
    P.S.v.m(——)
    {
        P p = new P();
        p.m1(); // parent
        C c = new C();
        c.m1(); // child
        P p1 = new C();
        p1.m1(); // child
    }
}
```

→ In overridding the method resolution always taks care by JVM based on runtime Object & in overridding reference type never play any role.

11/04/11

Regular Expressions

→ Any group of Strings according to a particular pattern is called "Regular Expression".

ex: ① We can write a Regular Expression to represent all valid mail-ids & By using that Regular Expression we can validate whether the given mail-id is valid or not.

② we can write a Regular Expression to represent all valid Java identifiers.

→ The main Application areas of Regular Expressions are

1. we can implement validation mechanism.
2. we can develope pattern matching applications.
3. we can develope translators like Compilers, interpreters e.t.c.
4. we can use for designing digital circuits
5. we can use to develope Communication protocols like TCP, IP, UDP e.t.c

ex:

```
import java.util.regex.*;
class RegExDemo
{
 p.s.v.m(String[] args)
 {
   Pattern p = Pattern.compile("ab");
   Master m = p.matcher("abbbabb cbdab");
```

```
while(m.find())
{
S.o.println(m.start() + "---" + m.end() + "---" + m.group());
}
}
```

o/p:-
```
0---2---ab
4---6---ab
10---12---ab
```

Pattern class:-

→ A pattern object represents Compiled version of regular Expression

We Can Create a pattern object by using compile() of pattern class.

```
Pattern p = Pattern.compile(String regularExpression);
```

Matcher Class:-

→ A matcher object Can be used to match character Sequence against a regular Expression. We Can Create a Matcher Object by using matcher() of pattern class

```
Matcher m = p.matcher(String target);
```

Important methods of matcher class:-

(1) boolean find();

→ It attempts to find next match & if it is available returns True otherwise returns false.

(ii) int start();-

→ returns start index of the match

(iii) int end();

→ returns end index of the match

(iv) String group();

→ returns the matched pattern

Character Classes :-

① [a-z] → Any lower case alphabet Symbol

② [A-Z] → Any upper " "

③ [a-zA-Z] → Any alphabet Symbol

④ [0-9] → Any digit from 0 to 9

⑤ [abc] → either a or b or c

⑥ [^abc] → Except a or b or c.

⑦ [0-9a-zA-Z] → Any alpha numeric character.

Ex:-

```
Pattern p = Pattern.compile("x");
Matcher m = p.matcher("a3b@c4z#");
while(m.find())
{
S.o.println(m.start() + "----" + m.group());
}
```

x = [ab]

0 ----a
2 ----b

x = [a-z]

0 ----a
2 ----b
4 ----c
6 ----z

x = [0-9]

1 ----3
5 ----4

x = [0-9a-z]

0 ----a
1 ----3
2 ----b
4 ----c
5 ----4
6 ----z

Predefined-character class :-

Space character ⟶ \s

[0-9] ⟶ \d

[0-9a-zA-Z] ⟶ \w

Anychaaracter ⟶ .

Ex:

```
Pattern p = Pattern.compile("x");
Matcher m = p.matcher("a3z4@ K7#");
                      0 1 2 3 4 5 6 7 8
while(m.find())
{
 S.o.println(m.start() + "----" + m.groups());
}
```

x = \\d	x = \\w	x = \\s	x = .
1 ---- 3	0 --- a	5 ----	0 — a
3 ---- 4	1 --- 3		1 — 3
7 ---- 7	2 --- z		2 — z
	3 --- 4		3 — 4
	6 --- k		4 — @
	7 --- 7		5 —
			6 — K
			7 — 7
			8 — #

Quantifiers :-

↳ We can use Quantifiers to specify no. of characters to match

Ex:

1) a ⟶ Exactly one a
2) a^+ ⟶ atleast one a
3) a^* ⟶ Any no. of a's
4) a? ⟶ atmost one a

ex:-

```
Pattern p = Pattern.compile("x");
Matcher m = p.matcher("abaabaaab");
while(m.find())
{
  S.o.pln(m.start()+"----"+m.group());
}
```

x=a	x=a+	x=a*	x=a?
0---a	0---a	0---a	0---a
2---a	2---aa	1-----	1----
3---a	5---aaa	2----aa	2----a
5---a		4-----	3---a
6---a		5----aaa	4----
7---a		8----	5----a
		9----	6----a
			7----a
			8----
			9-----

Split method (s) :-

Pattern class Contains split() method to split given String according to a Regular Expression.

ex:-

```
Pattern p = pattern.compile(" ");
String[] s = p.split("Durga Software Solutions");
for(String s1 : s)
{
  S.o.pln(s1); // Durga
}                Software
                 Solutions.
```

Ex(2):-

```
Pattern p = pattern.compile("\\.");        "[.]"
String[] s = p.split("www.durgajobs.com");
for(String s1 : s)
{
  S.o.println(s1);
}
```

o/p:-
www
durgajobs
com

String class split() method:-

→ String class also contains split() to split the given string against a regular expression

Ex:-

```
String s = "www.durgajobs.com";
String[] s1 = s.split("\\.");
for(String s2 : s)
{
  S.o.println(s2);
}
```

www
durgajobs
com

Note:-

Pattern class split() can take target string as argument where as String class split() can take regular expression as argument.

StringTokenizer :-

→ We can use StringTokenizer to divide the target String into Stream of Tokens according to the

→ StringTokenizer class presenting in java.util package.

Ex:-

① StringTokenizer st = new StringTokenizer("Durga Software Solutions");

```
while(st.hasMoreTokens())
{
 S.o.pln(st.nextToken());
}
```

o/p:-
Durga
Software
Solutions

Note:- The default regular Expression is Space

② StringTokenizer st = new StringTokenizer("1,00,000", ",");

```
while(st.hasMoreTokens())
{
 S.o.pln(st.nextToken());
}
```

o/p:-
1
00
000

o/p :-
1
00
000

Ex(1):- Write a Regular Expression to represent the Set of all valid identifiers in Java language.

Rules:- (1) the length of each identifier is atleast 2

(2) the allowed characters are a to z
A to Z
0 to 9
_

(3) the first character should not digit

R.E:- [a-zA-Z._][a-zA-Z0-9._]* [a-zA-Z0-9._]*

$x \cdot x^* = x^+$

[a-zA-Z._][a-zA-Z0-9._]+

```
import java.util.regex.*;
class RegExDemo2
{
  p.s.v.m(String[] args)
  {
    Pattern p = Pattern.compile("[a-zA-Z._][a-zA-Z0-9._]+");
    Matcher m = p.matcher(args[0]);
    if(m.find()&& m.group().equals(args[0]))
    {
      S.o.pln("Valid Identifier");
    }
    else
    {
      S.o.pln("Invalid Identifier");
    }
  }
}
```

2/8/21

② W.a. RE to represent all valid mobile numbers

Rule:- (1) mobile no contains 10 digits

(2) the first digit should be 7 to 9

RegEx:- [7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]

(or)

[7-9]\d{9}

③ W.a. Regular Expression to represent all valid mail_ids

Rules:

(1) The set of allowed characters in mail-id are 0 to 9, a-z, A-Z, ., -

(2) Should starts with alphabet Symbol

(3) Should contain atleast one Symbol.

RegExp:- [a-zA-Z][a-zA-Z0-9._-]*@[a-zA-Z0-9]+([.][a-zA-Z]+)+

RegExp:

" @gmail[.]com

" @(gmail|yahoo|hotmail)[.]com

Ex:-

```
import java.io.*;
import java.util.regex.*;
class MobileExtractor1
{
  P.S.V.m(String[] args) throws IOException
  {
    PrintWriter pw = new PrintWriter("mobile.txt");
    BufferedReader br = new BufferedReader(new FileReader("input.txt"));
```

```
String line = br.readLine();
Pattern p = Pattern.compile("[7-9][0-9]{9}");
while(line != null)
{
 Matcher m = p.matcher(line);
 while(m.find())
 {
  pw.println(m.group());
 }
  line = br.readLine();
 }
  pw.flush();
 }
}
```

P) W.a.p to Extract mail-ids from the given file where mail-ids are mixed with some raw data ?

→ In the above Example replace regular Expression with the following mail-id regular Expression.

$$[a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^{*}@[a\text{-}zA\text{-}Z0\text{-}9]^{+}([.][a\text{-}zA\text{-}Z]^{+})^{+}$$

P) W.a.p to display all text-files present in the given directory ?

```
import java.io.*;
import java.util.regex.*;
class FileNameExtractor
{
```

```
Public Static void main (String[] args) throws IOException
{
  int count = 0;
  Pattern p = pattern.compile("[a-zA-Z0-9_$]+[.]txt");
  File f = new File ("D:\\durga_classes");
  String[] s = f.list();
  for (String s1 : s)
    {
      Matcher m = p.matcher(s1);
      if(m.find() && m.group().equals(s1))
       {
         count++;
         S.o.pln(s1);
       }
    }
    S.o.pln (count);
  }
}
```

P) W.a.p to delete all .bak files present in D:\durgaclass

```
import java.io.*;
import java.util.regex.*;
class FileNamesDeleter
{
  P.S.v.m(String[] args) throws IOException
  {
    int count = 0;
    Pattern p = pattern.compile("[a-zA-Z0-9_$]+[.]bak");
```

```
File f = new File("D:\\durga_classes");
String[] s = f.list();
for(String s1 : s)
{
  Matcher m = p.matcher(s1);
  if(m.find() && m.group().equals(s1))
  {
    count++;
    S.o.println(s1);
    File f1 = new file(f,s1);
    f1.delete();
  }
}
S.o.println(count);
}
}
```

Enumeration (enum)

15/5/11

→ We can use enum to define a group of named constants

Ex!:① enum month
{
JAN, FEB, MAR ----- DEC(;) → optional.
}

② enum Bear
{
KF, KO, RC, FO(;) → optional
}

→ By using enum we can define our own data types

→ enum Concept introduced in 1.5v.

→ When compared with old languages enum Java's enum is more powerfull.

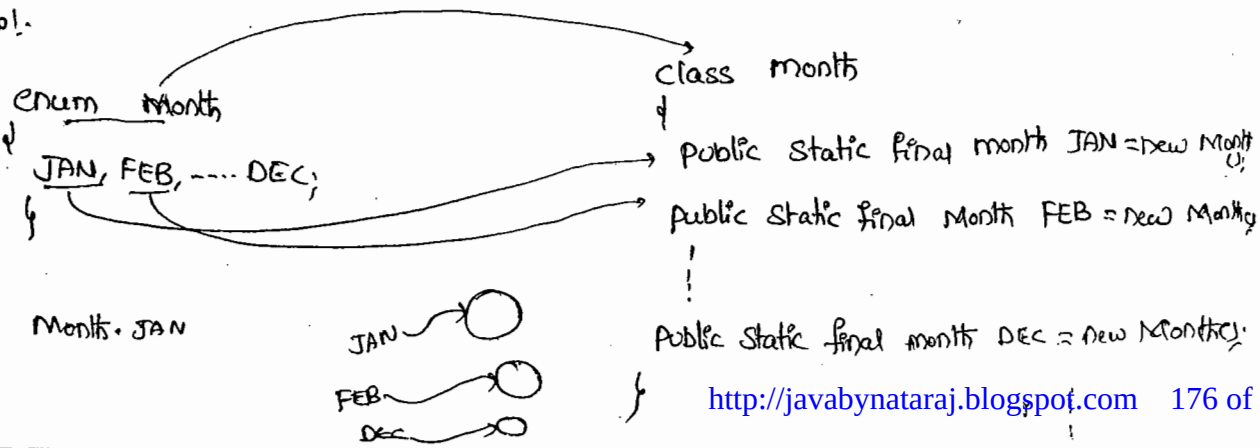
Internal implementation of enum :-

→ enum Concept internally implemented by using class Concept.

→ every enum constant is a reference variable to enum type object.

→ every enum constant is always public static final by default.

Eg!.



Declaration and usage of enum :-

Ex:-

```
enum Beer
{
 KF, KO, RC, FO;
}
Class Test
{
 P.S.V.m(String[] args)
 {
   Beer b1 = Beer.KF;
   S.o.pln(b1); // KF
 }
}
```

→ We can declare enum either with in the class or outside of the class but not inside a method.

→ If we are taying to declare enum with in a method we will get Compiletime Error.

Ex!-

```
enum X
{
}
Class Y
{
}
```
✓

```
Class Y
{
  enum X
  {
  }
}
```
✓

```
Class Y
{
  public void m1()
  {
    enum X
    {
    }
  }
}
```
✗

C.E:- enum types must not be local

→ if we declare enum outside the class the applicable modifiers are public, default, strictfp.

→ if we declare enum with in a class the applicable modifiers are public, default, strictfp, private, protected, static.

enum Vs Switch Statement :-

→ until 1.4v the allowed data-types for switch argument are byte, short, char, int.

→ But from 1.5v onwards in addition to above the corresponding wrapper classes (Byte, Short, Char, Integer,) + enum type also allowed

```
Switch (  )
{
}
```

1.4 V	1.5V	1.7V
byte short char int	Byte Short Character Integer enum	String

→ Hence from 1.5 version onwards we can use enum as argument to switch statement.

Ex:-

```
enum Beer
{
  KF, KO, RC, FO;
}
class Test
{
```

```
p.s.v.m(-->
{
 Beer b1 = Beer.Rc;
 switch (b1)
 {
  Case KF:
        S.o.pln(" It is children's brand);
        break;
  Case KO:
        S.o.pln(" It is too lite");
        break;
  Case Rc:
        S.o.pln(" It is challengers brand");
        break;
  Case FO:
        S.o.pln(" Buy one get one");
        break;
  default:
        S.o.pln(" other brands not recommended to
                                    take");
    }
  }
 }
```

Op:- It is challengers brand.

→ If we are passing enum type as argument to switch statement every case label should be a valid enum constant.

Eg:-
```
enum Beer
{
  KF, KO, RC, FO;
}
Beer b1 = Beer.KF;
Switch(b1)
{
  Case KF: ✓
  Case KO: ✓
  Case RC: ✓
  Case KALYANI: ✗ C.E:- UnQualified Enumeration Constant name required
}
}
```

enum Vs Inheritance :-

→ every enum in Java is direct child class of java.lang.Enum

→ As every enum is always extending java.lang.Enum there is no chance of extending any other enum (because Java won't provide support for multiple inheritance).

→ As every enum is always final implicitly we can't create child enum for our enums.

→ because of above reasons we can conclude inheritance concept is not applicable for enums Explicitly.

→ But enum can implement any no. of interfaces at a time.

Ex:- ①

```
enum x
{
}
enum y extends x
{
}
```

C.E:-

Cannot inherit from final x

enum types are not extensible

②

```
enum x extends java.lang.Enum
{
}
```

C.E:-

③

```
enum x
{
}
class y extends x
{
}
```

C.E1:- Cannot inherit from final x

C.E2:- enum types are not extensible

④

```
class x
{
}
enum y extends x
{
}
```

C.E:-

⑤

```
interface x
{
}
enum y implements x
{
}
```

java.lang.Enum :-

↦ every enum in java ~~should~~ is always direct child class of java.lang.Enum class.

↦ The power of enum is inheriting from this class only to our enum classes.

(Javap java.lang.Enum)

↦ It is an abstract class & direct child class of Object class.

↦ this class implements Comparable & Serializable Interfaces. hence every enum in java is bydefault Serializable and Comparable.

values() method :-

→ we can use values() method to list out all values of enum.

eg:- `Beer[] b = Beer.values();`

ordinal() method :-

→ with in the enum the order of constants is important we can specify its order by using ordinal value.

→ we can find ordinal value of enum constant by using ordinal method.

```
Public int ordinal();
```

→ Ordinal value is zero-based.

Eg:-

```
enum Beer
{
  KF, KO, RC, FO;
}
```

```
class Test
{
  p.s.v.m(String[] args)
  {
    Beer[] b = Beer.values();
    for(Beer b1 : b)
    {
      S.o.pln(b1+"----"+b1.ordinal());
    }
  }
}
```

o/p :-
KF ---- 0
KO ---- 1
RC ---- 2
FO ---- 3

Enum class Constructors & Speciality of Java enum :-

→ When Compared with old languages enum, Java enum is more Poweful because in addition to Constants we Can take Variables, methods, Constructors e.t.c... which may not possible in old languages. This extra facility is due to internal implementation of enum Concept which is class based.

→ Inside enum we Can declare main() method & hence we Can invoke enum class directly from Command prompt.

eg:- ① enum Fish
```
{
  STAR, GOLD, GUPPY, APOLLO, KILLER;   → mandatory.
  p.s.v.m(String[] args)
  {
    S.o.pln("ENUM MAIN METHOD");
  }
}
```

> Javac fish.java

> Java Fish

o/p!- Enum main method.

→ In addition to Constant if we want to take any extra members Compulsory List of Constants should be in the 1st Line & should ends with ";"

ex!- ① enum Color
{
RED, GREEN, BLUE;
Public void m1()
{
}
}

② enum Color
{
Public void m1()
{
}
RED, GREEN, BLUE;
}

③ enum Color
{
RED, GREEN, BLUE;
public void m1()
{
}
}

④ enum Color
{
Public void m1()
{
}
}

⑤ enum Color
{
}

→ Inside enum with out having Constant we can't to take any extra members, but Empty enum is always valid.

ex!.

enum Color
{
public void m1()
{
}
}

enum Color
{
}

Enum class Constructors :-

→ with-in Enum we can take Constructors also.

→ Enum class Constructors will be executed automatically at the time of Enum class loading. Hence because Enum Constants will be created at the time of class loading only.

→ We can't invoke Enum Constructors explicitly

Ex!-

```
enum Beer
{
  KF, KO, RC, FO;
  Beer()
  {
    S.o.pln("Constructor");
  }
}
Class Test
{
  Public static void main(String[] args)
  {
    Beer b1 = Beer.KF;
    S.o.pln(b1);
  }
}
```

o/p!-

```
Constructor
Constructor
Constructor
Constructor
KF
```

→ We Can't Create Objects of Enum explicitly & hence we Can't Call Constructors directly.

```
Beer b = new Beer();   X
```

C.E!-
Enum types may not be instantiated.

Ex:-
```
enum Beer
{
KF(75), KO(90), RC(90), FO;
int price;
Beer(int price)
{
this.price = price;
}
Beer()
{
this.price = 65;
}
Public int getprice()
{
return price;
}}
Class Test
{
p.s.v.m(String[] args)
{
Beer() b = Beer.values();
for(Beer b1 : b)
{
S.o.pln(b1 + "....." + b1.getprice());
}}}
```

KF ⇒ p.s.f. Beer KF = new Beer();

KF(100) ⇒ p.s.f. Beer KF = new Beer(100);

KF(100, "Gold", "Bitter")
⇒ p.s.f. Beer KF = new Beer(100, "Gold", "Bitter")

o/p!-
```
KF ..... 75
KO ..... 90
RC ..... 70
FO ..... 65
```

→ Within the enum we can take instance & static methods but we can't to take abstract methods

→ every enum constant represents an object hence what ever the methods we can apply on normal Java object we can apply those on enum constants also.

ex:-

Q) which of the following expressions are valid

✓ ① Beer.KF.equals(Beer.RC) // false

✓ ② Beer.KF.hashCode() // —

✓ ③ Beer.KF == Beer.RC ⟶ false

X ④ Beer.KF > Beer.RC

✓ ⑤ Beer.KF.ordinal > Beer.RC.ordinal

Case(1):-

```
Package pack1;
public enum Fish;
{
STAR, GUPPY, APOLLO;
}
```

```
Package pack2;
Class Test1
{
P.S.V.m(—)
{
S.o.pln(STAR);
}
}
```

```
import static pack1.Fish.STAR;
(or)
import static pack1.fish.*;
```

```
Package pack3;
--
Class Test2
{
 P.S.V.m(--)
 {
  Fish f = Fish.STAR;
  S.o.pln(f);
 }
}
```

```
import pack1.Fish;
    (or)
import pack1.*;
```

```
package pack4;
---
class Test3
{
 P.S.V.m(--)
 {
  Fish f = Fish.STAR;
  S.o.pln(Guppy);
 }
}
```

```
import pack1.Fish (or)
       import pack1.*;

import static pack1.Fish.Guppy;
          (or)
import static pack1.Fish.*;
```

Case2 :-

```
enum Color
{
  BLUE, RED
   {
     public void info()
     {
       S.o.pln(" Dangerous Color");
     }
   }, GREEN;

  public void info()
  {
   S.o.pln(" universal Color");
  }
}
```

```
class Test
{
  p.s.v.m(------>)
  {
    Color[] c = Color.values();
    for(Color c1 : c)
    {
      c1.Enfo();
    }
  }
}
```

O/P :- universal Color
Dangerous Color
universal Color.

Enum vs Enum vs Enumeration :-

Enum :-

→ It is a Keyword which can be used to define a group of named constants.

Enum :-

→ It is a class present in java.lang package which acts as a base class for all Java enums

Enumeration :-

→ It is an Interface present in java.util package, which can be used for retrieving objects from Collection one by one.

Internationalization (I18N)

I18N :-

→ various Countries follow various Conversions to represent dates & no.'s e.t.c

→ Our application should generate Locale Specific responses like for India people the response should be interms of Rs. (Rupees) & for the U.S people the response should be interms of dollars($) the process of designing such type of web application is called "Internationalization" (I18N).

→ We can implement I18N by using the following classes

① Locale
② NumberFormat
③ DateFormat

Locale :-

→ A Locale Object represents a Geo-graphic Location

Constructors :-

→ We can Create a Locale object by using the following Constructor.

(a) `Locale l = new Locale(String language);`

(b) `Locale l = new Locale(String language, String Country);`

javap java.util.Locale.

→ Locale class defines Several Constants to represent Some Standard Locales. We can use these Locale directly without Creating on our own.

Eg:- Locale.US Locale.ENGLISH
Locale.ITALIAN Locale.UK

Note:-

→ Locale class is the final class present in java.util package

→ It is the direct child class of Object it implements Cloneable & Serializable interfaces.

Important methods of Locale class:-

① public static Locale getDefault();

② public static void setDefault(Locale l);

③ public String getLanguage(); en

④ public String getDisplayLanguage(); english

⑤ public String getCountry(); us

⑥ public String getDisplayCountry(); unitedstates

⑦ public static String[] getISOCountries();

⑧ public static String[] getISOLanguages();

⑨ public static Locale[] getAvailableLocales();

Ex:-

```
import java.util.*;
Class LocaleDemo1
{
 Public static void main(String[] args)
 {
    Locale l1 = Locale.getDefault();
    S.o.pln(l1.getCountry() + "...." + l1.getLanguage());
    S.o.pln(l1.getDisplayCountry() + "----" + l1.getDisplayLanguage());
    Locale l2 = new Locale("pa","IN");
    Locale.setDefault(l2);
    String[] s3 = Locale.getISOLanguages();
    for(String s4 : s3)
    {
      S.o.pln(s4);
    }
    String[] s4 = Locale.getISOCountries();
     for(String s5 :s4)
     {
       S.o.pln(s5);
     }
     Locale[] s = Locale.getAvailableLocales();
     for(Locale s1:s)
     {
       S.o.pln(s1.getDisplayCountry() + "..." + s1.getDisplayLanguage());
     }
}}
```

NumberFormatt Classes :-

→ Various Countries follow various Conversions to represent Number by using NumberFormatt Class we Can formatt a numb according to a particular Locale.

→ NumberFormatt class present in java.text package & it is an abstract class. Hence we Can't Create NumberFormatt object directly

✗ NumberFormatt nf = new NumberFormatt();

Creating NumberFormatt object for the default Locale :-

→ NumberFormatt Class defines the following methods for this

① public static NumberFormatt getInstance();

② public static NumberFormat getCurrencyInstance();

③ public static NumberFormat getPercentInstance();

④ public static NumberFormat getNumberInstance();

Getting NumberFormat object for a Specific Locale :-

→ we have to pass the Corresponding Locale object as arguement to the above methods

Ex:
① public static NumberFormat getCurrencyInstance(Locale l);

→ Once we got NumberFormat Object we can format & parse numbers by using the following methods of NumberFormat class

① public String format(long l);

② public String format(double d);

→ To format (or) Convert java specific number-form to Locale Specific String form.

③ public Number parse(String s) throws ParseException

→ To Convert Locale Specific String form to java Specific Number form.

Ex1:-

w.a.p to represent a Java number in Italy Specific form.

```
import java.text.*;
import java.util.*;
class NumberFormatDemo2
{
  P.S.V.m(___)
  {
    double d=123456.789;
    NumberFormat nf=NumberFormat.getInstance(Locale.ITALY);
    S.o.pln("Italy form is:"+nf.format(d));
  }
}
```

o/p:- Italy form is: 123.456,789

Exp1:- w.a.p to represent a java number in India, U.K & U.S Currency forms.

```
import java.text.*;
import java.util.*;
Class NumberFormatDemo3
{
   P.S.V.m(------)
   {
      double d = 123456.789;
      Locale india = new Locale("pa","IN");   // any Location in India
      NumberFormat nf1 = NumberFormat.getCurrencyInstance(india);
      S.o.pln("India notation is...." + nf1.format(d));
      NumberFormat nf2 = NumberFormat.getCurrencyInstance(Locale.US);
      S.o.pln("US Notation is...." + nf2.format(d));
      NumberFormat nf3 = NumberFormat.getCurrencyInstance(Locale.UK);
      S.o.pln("UK notation is...." + nf3.format(d));
   }
}
```

O/P:- India Notation is ..... INR 123,456.79
US Notation is ..... $ 123,456.79
Uk Notation is ..... Ù 123,456.79

Setting Maximum & minimum integer & fraction digits :-

→ NumberFormat class defines the following methods to set maximum & minimum fraction & integer digits.

① public void setMaximumFractionDigits(int n);

② public void setMinimumFractionDigits(int n);

③ public void setMaximumIntegerDigits(int n);

④ public void setMinimumIntegerDigits(int n);

18/5/11

Ex1.

```
NF nf = NF.getInstance();

① nf.setMaximumFractionDigits(2);
    S.o.pln(nf.format(123.4567)); // 123.45
    S.o.pln(nf.format(123.4)); //123.4

② nf.setMinimumFractionDigits(2);
    S.o.pln(nf.format(123.4567)); // 123.4567
    S.o.pln(nf.format(123.4)); // 123.40

③ nf.setMaximumIntegerDigits(3);
    S.o.pln(nf.format(123456.234)); // 456.234
    S.o.pln(nf.format(12.3456)); // 12.3456

④ nf.setMinimumIntegerDigits(3);
    S.o.pln(nf.formatt(123456.234)); //456.234
    S.o.pln(nf.formatt(12.3456)); // 012.3456
```

Dateformat class :-

→ Various Countries follow various Conversions to represent Date.

→ By using Dateformatt class we can formatt the DATE according to a particular Locale.

→ DateFormat class is an abstract class & present in java.text pack.

Getting DateFormat Object for Default Locale :-

DateFormat class defines the following methods for this

(1) public Static Dateformat getInstance();

(2) public static Dateformat getDateInstance();

(3) public Static Dateformat getDateInstance(int Style);

→ DateFormat.Full → 0
DateFormat.LONG → 1
DateFormat.MEDIUM → 2
DateFormat.SHORT → 3

Getting Dateformat object for the Specific Locale :-

① Public Static DateFormat getDateInstance(int Style, Locale l);

→ Once we got DateFormat Object we can format & parse dates by using the following methods.

① public String format(Date d);

→ To Convert Java Date form to Locale Specific String form

Note:-
Default Style is Medium & most of the cases default Locale is US

② Public Date parse(String s) throws ParseException

To Convert Locale Specific Date Form to java DateForm.

Ex(1):-

W.a.p To display System Date in all possible Styles of U.S format

```
import java.util.*;
import java.text.*;
Class DateFormatDemo1
{
  p.s.v.m(_____)
  {
    S.o.pln("full form:" + DateFormat.getDateInstance(0).
                                    format(new Date()));
          (or)
  // DateFormat df = Dateformat.getDateInstance(0);
     S.o.pln(df.format(new Date());

  S.o.pln("Long form:" + DF.getDateInstance(1).format(new Date))
  S.o.pln("MEDIUM form:" + DF.getDateInstance(2).format(new Date()));
  S.o.pln("SHORT form:" + DF.getDateInstance(3).format(new Date()));
  }
}
```

O/P:- full form: Thursday, february. 2. 2010

Long form : February 18, 2010

medium form : Feb 18, 2010

short form : 2/18/10

Ex 2).

(P) w.a.p to display System Date US, Uk & Italy form.

```
{
    S.o.pln("US form:" + DF.getDateInstance(0, Locale.US).format(
                                                    new Date()));
    S.o.pln("Uk-form:" + DF.getDateInstance(0, Locale.UK).format(new Date()));
    S.o.pln("ITALY form:" + DF.getDateInstance(0, Locale.ITALY).format(new Date()));
}
```

o/p:-

US form : Tuesday, May 18, 2010

Uk form : Tuesday, 18 May 2010

ITALY form: martedo 18 maggie 2010

Getting Dateformat object to represent both DATE & TIME :-

① public static Dateformat getDateTimeInstance();

② public static Dateformat getDateTimeInstance(int datestyle, int timestyle);

③ public static Dateformat getDateTimeInstance(int datestyle, int timestyle, Locale);

Ex!.

```
S.o.pln("US form:" + DateFormat.getDateTimeInstance(0, 0, Locale.US)
                                                    .format(new Date()));
```

o/p:-

US form : Tuesday, May 18, 2011 9:53:45 AM GMT:+5:30

Note:- Default style is medium & most of the cases default Locale is US

Development

javac :-

We can use this Command to Compile a Single or group of .java files.

Syn:- javac [options] A.java/

A.java B.java

*.java

[options]
- -d
- -Source
- -cp
- -classpath
- -version

java :-

We can use java Command to run a .class file

Syn:- java [options] A ↵

[options]
- -ea | -esa | -da | -dsa
- -version
- -cp / -classpath
- -D
- :

Note:- We can Compile a group of .java files at a time where as we can run only on .class file at a time.

Classpath :-

→ Classpath describes the location where required .class files are available.

→ JVM will always use Classpath to locate the required .class file.

→ The following are various possible ways to Set the Classpath.

① Permanently by using Environment variable classpath.

→ This classpath will be preserved after system restart also

② At Command prompt level by using Set Command.

Set classpath = %classpath%;D:\path>

→ This classpath will be applicable only for that particular Comand prompt window only. Once we close that Command prompt automatically classpath will be lost

③ At Command Level by using -cp option

java -cp D:\path> Test ↵

→ This classpath is applicable only for this particular Command. Once Command execution Completes automatically classpath will be lost.

* Among the above 3 ways the most Commonly used approach is Setting classpath at Command Level.

Ex!- class Test
{
p.s.v.m(——)
{
S.o.pln("Classpath Demo");
}
}

D:\Durgaclasses\> javac Test.java ↵
>java Test ↵

o/p!- Classpath Demo

✗ D:\java Test ↵ R.E!- NoClassDeffoundError

✓ D:\> java -cp D:\Durgaclasses Test ↵ ✓

o/p!- ClasspathDemo

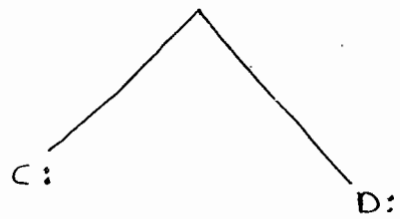
✓ D:\> java -cp D:\Durgaclasses Test ↵

o/p!- ClasspathDemo

Note:-

If we set classpath explicitly then we can run Java program from any location but if we are not setting the classpath then we have to run java program only from current working directory.

Ex:-




```
public class fresher
{
  public void m1()
  {
    S.o.pln ("I want JOB");
  }
}
```

```
class Company
{
  p.s.v.m (---->)
  {
    fresher f = new fresher();
    f.m1()
    S.o.pln("Getting JOB is very
             easy.. Not required to
             worry");
  }
}
```

C:\> javac fresher.java ✓

D:\> javac Company.java ✗

C.E:- Cannot find symbol
Symbol: class fresher
location: class Company

D:\> javac -cp C: Company.java ✓

✗ D:\javac Company ←

R.E:- NoClassDefFoundError: fresher

✗ D:\java -cp C: Company ←

R.E:- NoClassDefFoundError: Company

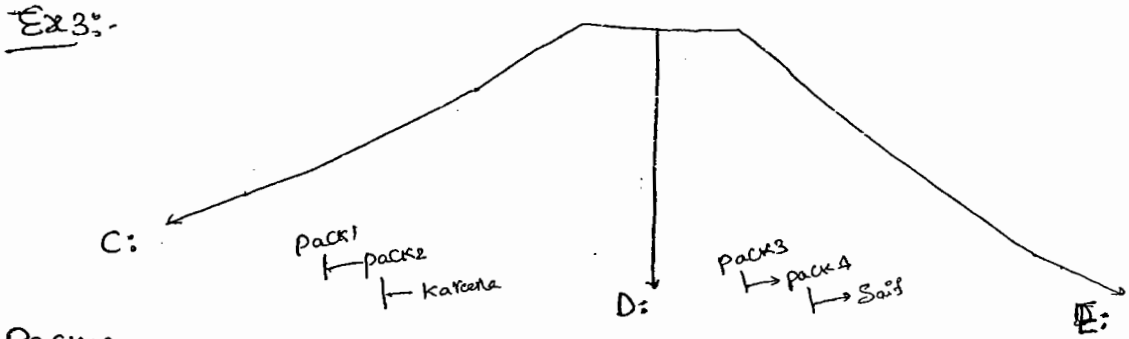
✓ D:\ java -cp D:;C: Company (or) D:\java -cp .;C: Company

o/p :- I can JOB

Getting JOB is very easy... not required to worry.

✓ E:\ java -cp D:;C: Company

Ex3:-




```
Package pack1.pack2;
Public class Kareena
{
 public void m1()
 {
S.o.pln(" Hello Saif Can I help U");
 }
}
```

```
package pack3.pack4;
Import pack1.pack2.Kareena
Public class Saif
{
Public void m1()
{
Kareena k = new Kareena();
K.m1();
S.o.pln(" Not possible.. As I am in SCJP class");
}
}
```

```
Import pack3.pack4.Saif;
class Durga
{
 p.s.v.m(—)
 {
 Saif s = new Saif();
 S.m1();
 S.o.pln(" Can please tell hello —func");
 }
}
```

✓ C:\> java -d. Kareena

✓ D:\> java -d. Saif.java

C.E:- Cannot find Symbol

Symbol : class Kareena

Location : class Saif

✓ D:\ java -cp c: -d . Saif.java

✗ E:\ javac Durga.java

C.E:- Cannot find Symbol

Symbol : class Saif

location : class Durga

✓ E:\> javac -cp D: Durga.java

✗ E:\> java Durga ↵

R.E:- NoClassDefFoundError : Saif

✗ E:\> java -cp D: Durga ↵

R.E:- NoClassDefFoundError : Durga

✗ E:\> java -cp .;D: Durga

R.E:- NoClassDefFoundError : Kasreena

✓ E:\> java -cp E:;D:;C: Durga

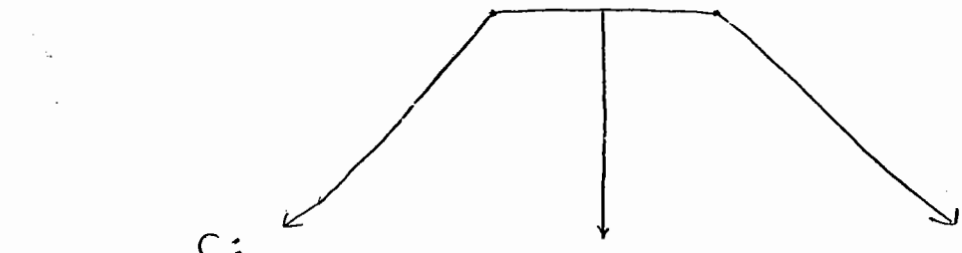
Note:-

① Compiler will check only one level dependency where as JVM will checks all levels of dependency

② If any folder structure created because of package statement it should be resolved through import statement only. & Base package location we have to update in classpath.

③ within the classpath the order of locations is very important for the required .class file, JVM will always search the locations from

Left → Right in classpath. Once JVM finds the required file then the rest of the classpath won't be searched.




```
Class Nagavali
{
 p.s.v.m(--)
 {
 S.o.pln("C: Nagavali");
 }
}
```

```
Class Nagavali
{
 p.s.v.m(--)
 {
 S.o.pln("D: Nagavali");
 }
}
```

```
Class Nagavali
{
 p.s.v.m()
 {
 S.o.pln("E: Nagavali
 }
}
```

C:\> javac Nagavali.java ✓

D:\> javac Nagavali.java ✓

E:\> javac Nagavali.java ✓

C:\> java Nagavali ✓

o/p C: Nagavali

D:\> java -cp C:;D:;E: Nagavali ↵

o/p C: Nagavali

D:\> java -cp E:;D:;C: Nagavali ↵

o/p: E: Nagavali

D:\> java -cp D:;E:;C: Nagavali ↵

o/p: D: Nagavali

JAR file :-

→ If Several dependent files are available then it is never recommended to set each class file individually in the classpath we have to group all those .Class file into a Single Zip file. & we have to make that Zip file available in the class path. This zip file is nothing but JAR file.

Ex(1) :-

To develope Servlet all required .class files are available in Servlet-api.jar. we have to make this Jar file available in the classpath then only Servlet will be Compiled.

jar Vs war Vs ear :-

① jar : (java archieve file)

→ It Contains a group of .class files

② war :- (web archieve file)

→ It represents a web application which may Contain Servlets, Jsps, HTMLs, css file, JavaScripts, e.t.c..

③ ear :- (Enterprise archeive file)

→ It represents an enterprise application which may Contains Servlets, Jsps, EJBs, JNS Components e.t.c.

Various Commands :-

① To Create a jar file :-

```
jar -cvf durga.jar A.class B.class C.class
                   *.class
```

② TO extract a jar file :-

```
jar -xvf durga.jar
```

③ TO Display table of contents of a jar file :-

```
jar -tvf durga.jar
```

Ex :-

```
public class DurgaColorfullCalc
{
 public static int add(int x, int y)
 {
  return x*y;
 }
public static int add(int x, int y)
 {
  return 2*x*y;
 }
}
```

```
C:\> javac DurgaColorfullCalc.java
C:\> jar -cvf durgaCalc.jar DurgaColorfullCalc.class
```

```
class Bakasa
{
 p.s.v.m(——)
 {
 s.o.pln(DurgaColorfulCalc.add(10,20));
 s.o.pln(DurgaColorfulCalc.multiply(10,20));
 }
}
```

✗ D:\> javac Bakasa.java

✗ D:\> javac -cp c: Bakasa.java

✓ D:\> javac -cp c:\durgaCalc.jar Bakasa.java

✓ D:\> javac -cp .;c:\durgaCalc.jar Bakasa.

o/p:- 200
400

Note:-

→ when ever we are placing a jar file in the classpath Compulsary name of the jar file we should include, just Location is not enough.

Shortcut way to place jar file :-

→ If we are placing the jar file in the following Location then it is not required to set classpath explecitly bydefault it is available to Jvm & Java Compiler.

JDK
↳ JRE
↳ Lib
↳ ext
↳ *.jar

System properties :-

→ For every System persistance information will be maintain in the form of System properties. These may include O.S name, Varchual machine version, User Country .e.t.c....

→ We can get System properties by using getProperties() method of System class

Ex!- Demo program to print all System properties.

```
import java.util.*;
class Test
{
  public static void main(String[] args)
  {
    properties p = System.getProperties();
    p.list(System.out);
  }
}
```

→ We can set System property from the Command prompt by using -D option

Ex!- Java -Ddurga=SCJP Test

Space is not allowed

durga → name of the property

SCJP → Value of the property

9) JDK vs JRE vs JVM :-

JDK:(Java developement kit) :-

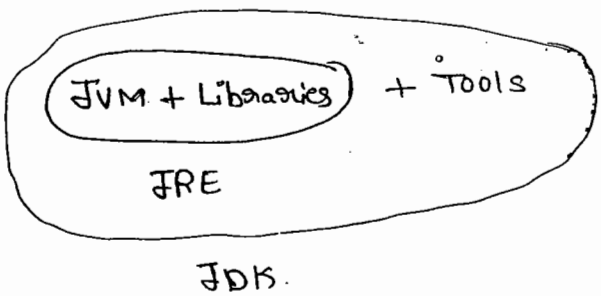
→ To develop & run Java application the required environment provided by JDK.

JRE:- (Java Runtime Environment) :-

→ To run Java application the required environment provided by JRE

JVM :-

→ this machine is responsible to execute Java program.



JDK = JRE + Tools

JRE = JVM + Libraries.

Note:-

→ On client machine we have to install JRE, where as on the developers machine we have to install JDK.

diff: b/w path & classpath :-

→ We can use classpath to describe the location where required .class files are available.

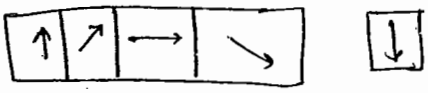
→ If we are not setting the classpath then our program won't be run.

Path :-

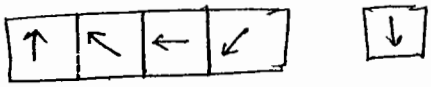
→ We can use path variable to describe the Location where required Binary executables are available.

→ If we are not setting path variable then java & javac Commands won't work.

3) Clockwise

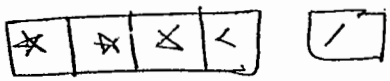


9) Anticlock

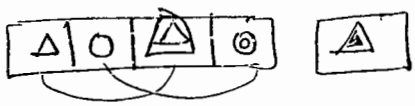


(3) Increasing

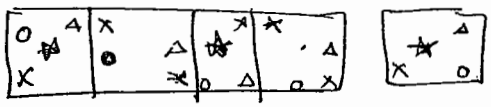
(4) Decreasing



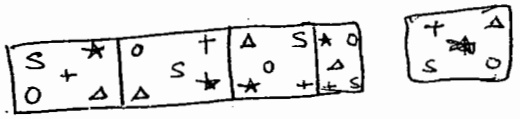
(5) Alternate



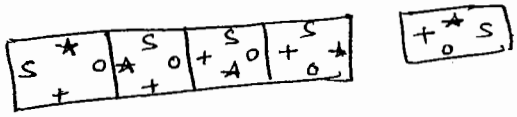
(6) multiple movement=



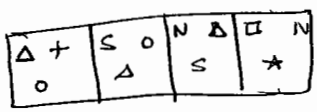
(7) Rotation



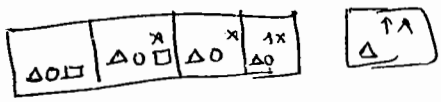
(8) Interchange

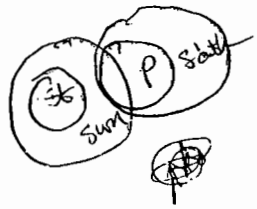


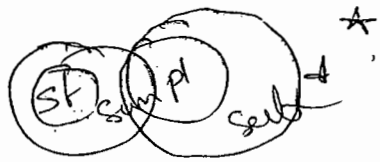
(9) missing of fig

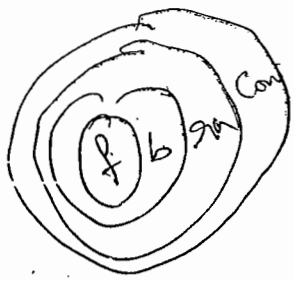


(10) Substitution









miracle

1.5V
- → Autoboxing & unboxing –
- → generics;
- → var-arg –
- → for-each
- → enum
- → Annotations ✓
- → Queue ✓
- → Static import } not recommended ✓
- → Co-varic of return types.

walk
↓
Jogging
↓
Running
↓
Sprinting

Siddhartha (VNR VJIT)
9951884313
Siddharthahprs@yahoo.co.in

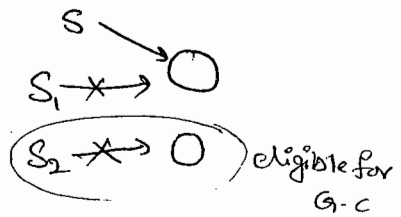
Vishnuteja.Y.S (
9703340473, 9493410648
Vishnuteja87@gmail.com

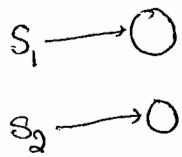
Vasu – 8179969444 (CEVMG)
slvudsarma@gmail.com

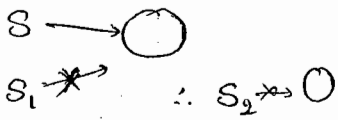
Ex 2:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  Student s = m1();
 }
 p.s.Student m1()
 {
  Student s1 = new Student();
  Student s2 = new Student();
  return s1;
 }
}
```

one object eligible for G.C →



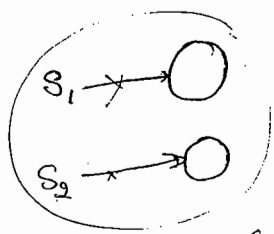




Ex 3 :-

```
Class Test
{
 p.s.v.main(String[] args)
 {
  m1();
 }
 p.s.Student m1()
 {
  Student s1 = new Student();
  Student s2 = new Student();
  return s1;
 }
}
```

Two objects eligible for G.C →

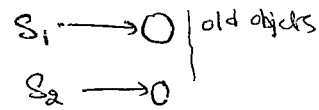


2) Reassigning the Reference variable :-

→ If an object is no longer required then reassign its reference variables to some other objects then that old object automatically eligible for G.C.

Ex(1) :-

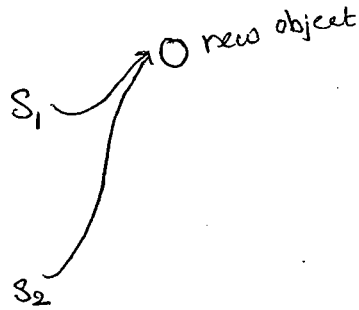
```
Student S1 = new Student();
Student S2 = new Student();
```



no object eligible for G.C →

```
S1 = new Student();
```

One object eligible for G.C →



Two objects eligible →

```
S2 = S1;
```

(3) Objects Created Inside a method :-

→ The Objects which are Created inside a method are by default eligible for G.C after Completing that method.

Ex :-

```
class Test
{
  p.s.v.main(String [] args)
  {
    m1();
  }
  p.s.v.m1()
  {
    Student S1 = new Student();
    Student S2 = new Student();
  }
}
```

2 objects eligible for G.C →