COLLECTIONS….

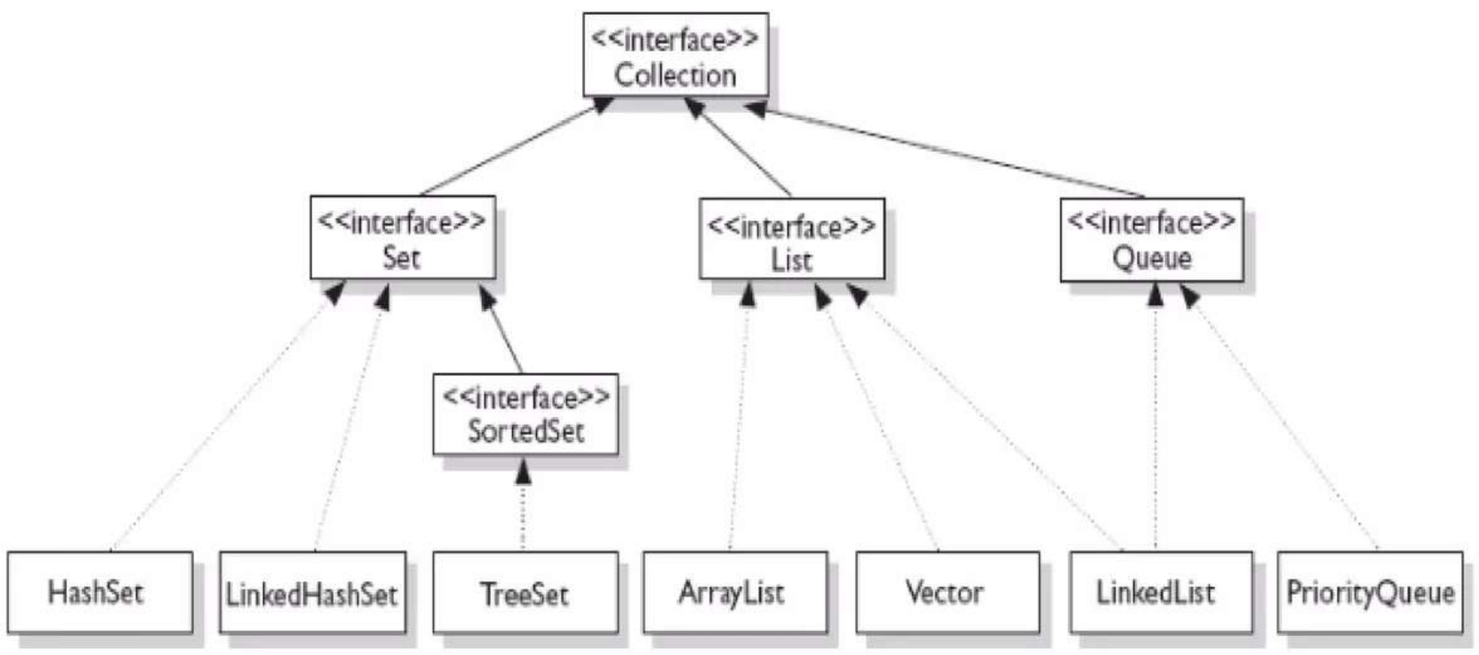
- **Collection** → Group of objects
  → To represent a group of "individual objects" as a single entity.
  → Root interface of entire collection framework.

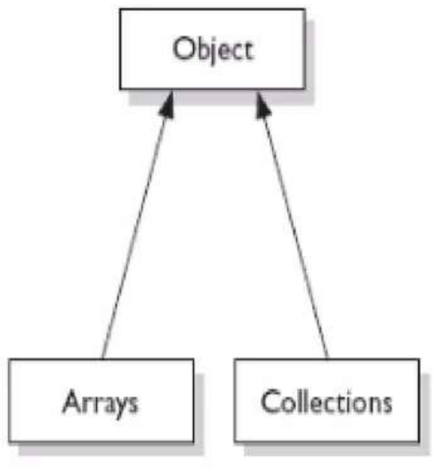
  - There is no concrete class which implements Collection interface directly.

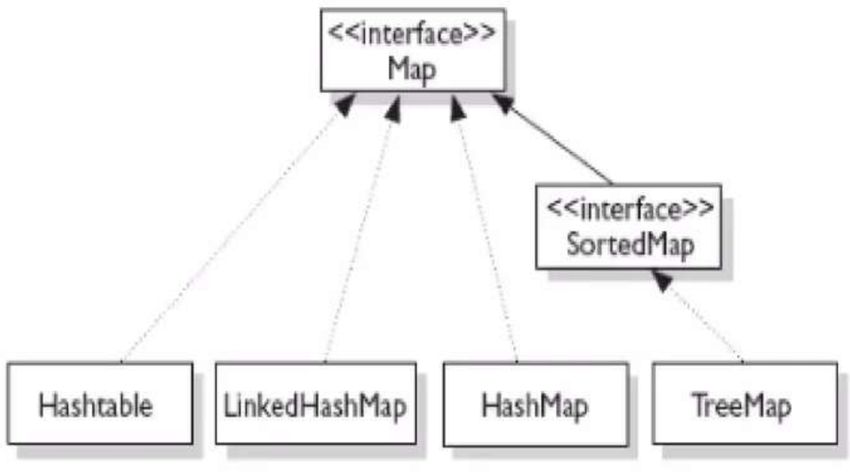
  - ***Consists of 3 parts*** :
    1. Core Interfaces.
    2. Concrete Implementation
    3. Algorithms such as searching and sorting.

- **ADVANTAGES →**
  - ✓ Reduces programming Effort.
  - ✓ Increases performance.
  - ✓ Provides interoperability between unrelated APIs
  - ✓ Reduces the effort required to learn APIs
  - ✓ Reduces the effort required to design and implement APIs
  - ✓ Fosters Software reuse.

- **Remember Interface / Class & Child Class →**







Arrays v/s Collections ?

Arrays	Collections
1) Fixed in size.	1) Growable in nature.
2) Memory point of view → Not Recommend	2) Highly recommended to use.
3) Performance point of view → Recommend	3) Not recommended to use.
4) Hold only Homogeneous data type element	4) Hold both Homo & Heterogeneous data types.
5) Hold both primitives & object types.	5) Holds only Objects but not primitives.

9 Key Collection Interface framework →

1. **Collection** - Collection interface defines the most common methods which can be applicable for any collection object.

2. **List** [child interface of Collection] - Represent group of individual objects as single entity where "*Duplicates are allow & Insertion order must be preserved*".

3. **Set** [child interface of Collection] - Maintain unique ele. "*Duplicates not allow & insertion order is not preserved*".

4. **SortedSet** [child interface of Set] - Sets that maintain elements in sorted order. "*Duplicates not allow but all objects insertion according to some sorting order*".

5. **NavigableSet** [child interface of SortedSet] - Defines several methods for navigation purposes.

6. **Queue** [child interface of Collection] - Objects *arranged in order* in which they are to be *processed.*

7. **Map** [NOT child interface of Collection] - Represent *group of objects* as key-value pairs. "*Duplicate keys not allowed but values can be duplicated*".

8. **SortedMap** [child interface of Map] - Represent *group of object* as key value pair "*according to some sorting order of keys*".

9. **NavigableMap** [child interface of SortedMap] - Defines several methods for navigation purposes.

- Interfaces (Collection, List, Set, SortedSet, NavigableSet, and Queue) meant for representing *a group of individual objects.*

By- Shalabh Chaudhary [GLAU]

Collection Implementations →

✓ Classes that implement collection interfaces have names in form - *<Implementation-style><Interface>*.

➢ **HashSet** - is an unSorted, unOrdered Set.
- Uses hashcode of object being inserted, More efficient your hashCode() implementation the better access performance you'll get.
- Use this class when you want a collection with no duplicates and you don't care about the order, when you iterate through it

➢ **LinkedHashSet** - Ordered version of **HashSet** that maintains ele doubly-linked List
- Used when you want to Iterate through the elements in the order in which they were inserted which is not possible in HashSet.

➢ **TreeSet** – Strores objects in sorted order by storing elements in tree.
- Can add only similar types of elements to tree set.

➢ **ArrayList** – Like Growable array which give Fast Iteration & Fast Random Access.
- It is an ordered collection(by index), but not sorted.
- ArrayList now implement new RandomAccess interface - Marker interface

➢ **LinkedList** - Is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.

Collection and Collections ?

✓ Collection – **Interface,** used to represent group of objects as single entity.
✓ Collections – **Utility class**, present in java.util package to define several utility methods for collection objects.

Collection Interface Methods →**[No concrete class to implement interface directly]**

1. Int size(); - Return no. of elements in collection.
2. boolean add(Object ele); - Add ele to invoking collection.
3. boolean contains(Object ele); - Return true if element is ele of invoking collection.
4. Iterator iterator(); - Returns iterator from invoking collection.
5. boolean remove(Object ele); - Remove one instance of element from invoking collection
6. boolean addAll(Collection c); - Add all element of c to invoking collection.
7. boolean containsAll(Collection c); - Return true if invoking collection contains all elements of c, else false.
8. boolean removeAll(Collection c); - Remove all element of c from invoking collection.
9. boolean retainAll(Collection c); - Remove all objects/ele except those present in c.
10. void clear(); - Remove all ele from invoking collection.
11. boolean isEmpty(); - Returns true if invoking collection is empty.
12. Object[] toArray(); - Return array that contains all ele stored in invoking collection.

Difference between HashMap and Hashtable →

Hashtable	HashMap
Hashtable class is synchronized.	HashMap is not synchronized.
Slower than HashMap, Because of Thread-safe.	Works Faster.
Neither **key** nor **values** can be null	Both **key** and **values** can be null
Order of table remain constant over time.	Doesn't guarantee that order of map will remain constant over time.

QQ. Which of the following Classes is NOT implemented from the Collection interface?
a. TreeSet
b. HashTable (Correct)
c. Vector
d. LinkedList

QQ. Which does NOT implement the Collection interface?
a.List
b.Map (Correct)
c.Set

QQ. Which of the following does NOT accept duplicate values?
a. ArrayList
b. LinkedList
c. TreeSet - Not Allow duplicate ele.
d. Vector

- **LIST Interface** [Extends from Collection Interface] →
  - ✓ It stores elements in a sequential manner.
  - ✓ Can contain duplicate elements.

➢ IMPORTANT METHODS →

1. boolean add(int index,Object o);
2. boolean addAll(int index,Collectio c);
3. Object get(int index);
4. Object remove(int index);
5. Object set(int index,Object new);//to replace
6. Int indexOf(Object o);
7. Returns index of first occurrence of "o".
8. Int lastIndexOf(Object o);
9. ListIterator listIterator();

- **ArrayList** [ArrayList class implements List interface] →
  - ✓ Supports dynamic array that can grow dynamically.
  - ✓ Can contain duplicate elements.
  - ✓ Heterogeneous objects are allowed.(Except TreeSet & TreeMap)
  - ✓ Insertion order preserved & Provides more powerful insertion and search mechanisms than arrays.
  - ✓ Null insertion is possible.

Syntax:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, newInteger(42));
int total=list.get(0).intValue();
```

❖ **Eg.:**

```
import java.util.*;
public class ArrayListTest{
    public static void main(String[] args) {
        ArrayList<String> test = new ArrayList<String>();
        String s = "hi";
        test.add("hello");
        test.add(s);
        test.add(s+s);
        System.out.println(test);      // [hello, hi, hihi]
        System.out.println(test.size());    // 3
        System.out.println(test.contains(42));  // false
        System.out.println(test.contains("hihi"));  // true
        test.remove("hi");
        System.out.println(test.size());    // 2
    }
}
```

- **Differences between ArrayList and Vector ?**

ArrayList	Vector
1) No method is synchronized	1) Every method is synchronized
2) At a time multiple Threads are allow to operate on ArrayList object and hence ArrayList object is not Thread safe.	2) At a time only one Thread is allow to operate on Vector object and hence Vector object is Thread safe.
3) Relatively performance is high because Threads are not required to wait.	3) Relatively performance is low because Threads are required to wait.
4) It is non legacy and introduced in 1.2v	4) It is legacy and introduced in 1.0v

LinkedList →

- ✓ Usaually used to Implement List and also the Stack(LIFO) & Queue(FIFO).
- ✓ Frequent Insertion / deletion in *Middle* then LinkedList - Best choice.
- ✓ Insertion order is Preserved.
- ✓ Null Insertion – Possible.
- ✓ Frequent Retrieval operation then LinkedList - Worst choice.
- ✓ Allow Duplicate objects.
- ✓ Allow Heterogeneous objects.
- ✓ Implements Serializable and Cloneable interfaces but not RandomAccess.

➢ **Constructors:**

- ✓ Creates an empty LinkedList object.

```
LinkedList l=new LinkedList();
```

- ✓ To create an equivalent LinkedList object for the given collection.

```
LinkedList l=new LinkedList(Collection c);
```

➢ **LinkedList IMPORTANT METHODS:**

- ✓ void addFirst(Object x)
- ✓ void addLast(Object x)
- ✓ Object getFirst()
- ✓ Object getLast()
- ✓ Object removeFirst()
- ✓ Object removeLast()

Vector Class →

- ✓ Vector data struct.- resizable / growable array implement by java.util.Vector class
- ✓ Same as ArrayList, but Vector methods are synchronized for thread safety.
- ✓ Insertion order is preserved.
- ✓ Null insertion is possible.
- ✓ Allow Duplicate objects.
- ✓ Allow Heterogeneous objects.
- ✓ Implements Serializable, Cloneable and RandomAccess interfaces.
- ✓ New java.util.Vector is implemented from List Interface.

➢ **Creation of a Vector -**

```
Vector v = new Vector(Collection c);   // allows old or new methods
List lv1 = new Vector();               // allows only the new (List) methods.
```

Stack →

- ✓ Child Class of Vector.
- ✓ Require Last In First Out(LIFO) order then - Use Stack.

➢ **Constructor:**

Contains only one constructor.

```
Stack s= new Stack();
```

- **IMPORTANT METHODS:**
  - ✓ Object push(Object o); // To insert an object into the stack.
  - ✓ Object pop(); // To remove and return top of the stack.
  - ✓ Object peek(); // To return top of the stack without removal.
  - ✓ boolean empty(); // Returns true if Stack is empty.
  - ✓ Int search(Object o); // Returns offset if element available else -1.

The 3 cursors of java →

✓ *To get objects one by one from the collection then - use cursor.*

1. **Enumeration**
2. **Iterator**
3. **ListIterator**

1,2. Enumeration & Iterator →

Java provides 2 interfaces that *define the methods by which you can access each element of collection.*

1. Enumeration - [Legacy – Yes **]**

- ✓ Is a legacy interface and is considered obsolete for new code.
- ✓ **elements()** method to Create Enumeration object -

```
public Enumeration elements();
Enumeration e=v.elements();   // v - Vector Object, Vector v=new Vector();
```

2. Iterator - [Legacy – No **]**

- ✓ Returns an iterator to a collection.
- ✓ It's an object that enables to traverse through collection.
- ✓ Can be used to remove elements from collection selectively.
- ✓ **iterator()** method of Collection interface to create Iterator object.

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

Enumeration() Methods →

1. hasMoreElements()
2. nextElement()
3. (Not Available)

Iterator Methods →

1. hasNext() - Returns true if more ele exist.
2. next() - Returns next element.
3. remove() - Removes current element.

1. Enumeration	2. Iterator
1. Can't do any modifications to collection while traversing.	Can remove element of collection while traversing it.
2. Introduced in JDK 1.0	Introduced from JDK 1.2
3. Used to traverse legacy classes like Vector, Stack & HashTable.	Used to iterate most of classes in collection framework like ArrayList, HashSet, HashMap, LinkedList etc.
4. Not safe and secured due to it's fail-safe nature.	Iterator is safer and secured than the Enumeration.

- **Limitations of Iterator:**
  1. Both enumeration and Iterator are single direction cursors only. i.e., can move only forward direction and not in backward direction.
  2. Can perform Read and Remove operations but can't perform replacement and addition of new objects.

*So, To overcome these limitations **ListIterator** concept introduced.*

3. ListIterator - **[** Legacy – No **]** [Child Interface of Iterator] (bi-directional cursor)

- ✓ Used for obtaining a iterator for collections that implement List.
- ✓ Gives ability to access the collection in either forward or backward direction.
- ✓ Has both next() and previous() method to access next & previous element in List.

❖ Eg.:

```
import java.util.*;
public class ListIteratorTest {
        public static void main(String[]args){
           ArrayList aList=new ArrayList();  //Add ele to ArrayList object
           aList.add("1");
           aList.add("2");
           aList.add("3");
           ListIterator listIterator=aList.listIterator();
           System.out.println("Prev Indx is:" +listIterator.previousIndex());
           System.out.println("Next Index is:"+listIterator.nextIndex());
           listIterator.next();        // increasing current position by one ele
           System.out.println("Prev Indx is: " +listIterator.previousIndex());
           System.out.println("Next Index is:"+listIterator.nextIndex());
        }
   }
```

- **Advantage of Iterator over for-each(**Also used for iterating**) method →**
  - ✓ To remove current element.
  - ✓ The for-each construct hides the iterator,so you cannot call remove
  - ✓ Iterate over multiple collections in parallel.

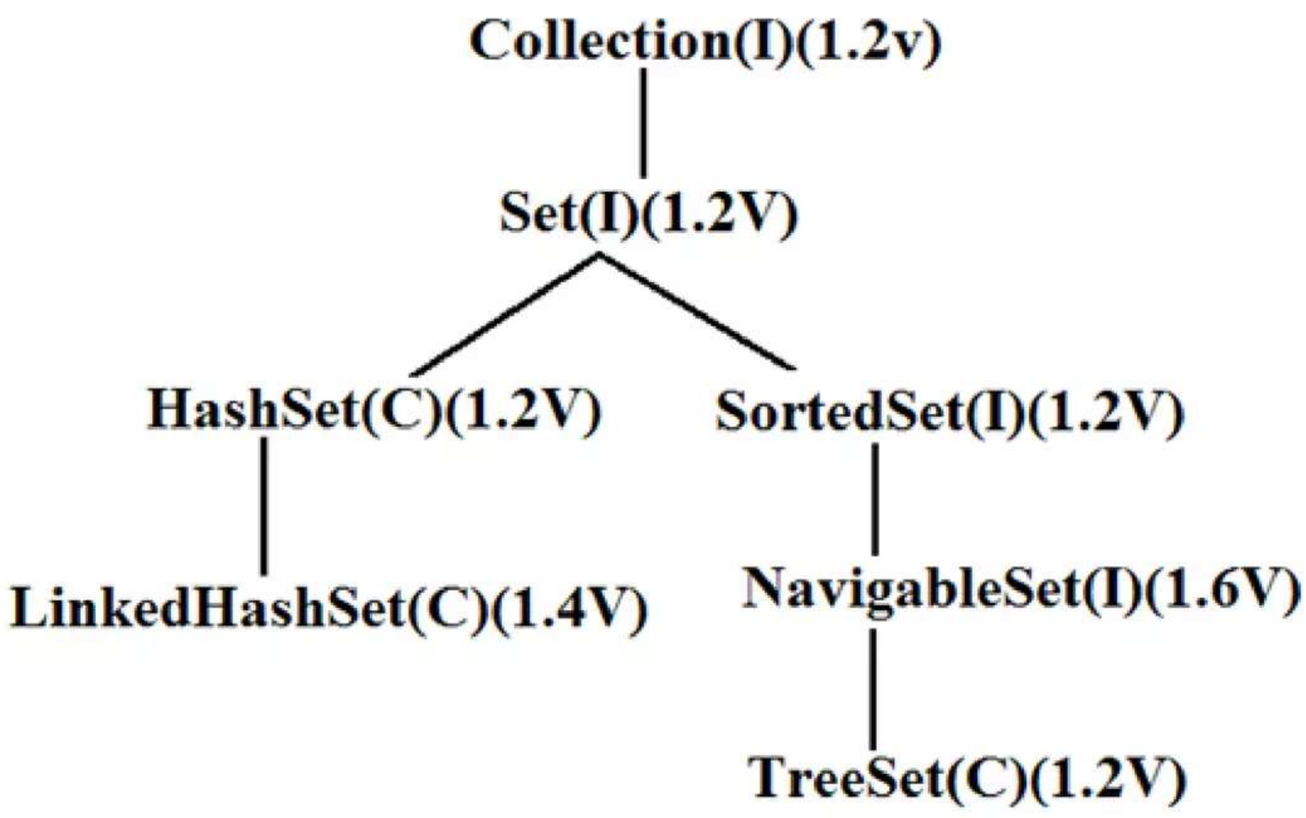
```
for(Object o : oa) {            // for-each construct
    Fruit d2= (Fruit)o;
    System.out.println(d2.name);
}
```

- **IMPORTANT POINTS** →
  - ✓ Use List to perform insert, delete and update operations based on position in list.
  - ✓ ArrayList, LinkedList & Vector implements Listinterface.
  - ✓ For Insertion & Deletion operation - Use LinkedList – Better Performance.
  - ✓ For Search operations – Use ArrayList – Faster.
  - ✓ ArrayList & LinkedList - NOT synchronized.
  - ✓ Vector – synchronized.
  - ✓ If thread safety not important - Use Either ArrayList or LinkedList.

QQ. In which of the following classes position based operations can be performed?

a. ArrayList
b. LinkedList
c. Vector
d. All of the above (correct)

- **Set Interface** →
  - ✓ Child Interface of Collection.
  - ✓ To represent a group of individual objects as a single entity.
  - ✓ Insertion order - NOT Preserved.
  - ✓ Duplicates Objects - NOT Allowed.
  - ✓ Use Collection Interface methods.



HashSet	LinkedHashSet
1) Underlying data structure is Hashtable.	1) Combination of LinkedList and Hashtable.
2) Insertion order - NOT Preserved.	2) Insertion order - Preserved.
3) Introduced in 1.2 v.	3) Introduced in 1.4v.
4) Duplicate objects – NOT Allowed.	4) Same
5) Heterogeneous Obj. – Allowed.	5) Same
6) Null insertion - Possible.(only once)	6) Same
7) Suitable if Frequent operation - Search	

Note: **LinkedHashSet** and **LinkedHashMap** - For implementing "Cache Applications" where Insertion order - Must be Preserved & Duplicates - NOT Allowed.

Comparable	Comparator
1) For default Natural sorting order.	1) For Customized sorting order.
2) Present in java.lang package.	2) Present in java.util package.
3) Contains only 1 method compareTo()	3) Contains 2 methods Compare() & Equals().
4) String class and all wrapper Classes implements Comparable interface.	4) Only implemented classes of Comparator are Collator & RuleBasedCollator. (used in GUI)

IMPORTANT →

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying DS	Hashtable.	LinkedList +Hashtable	Balanced Tree.
2) Insertion order.	Not preserved.	Preserved.	Not preserved (by default).
3) Duplicate objects.	Not allowed.	Not allowed.	Not allowed.
4) Sorting order.	Not applicable	Not applicable.	Applicable.
5) Heterogeneous obj.	Allowed.	Allowed.	Not allowed.
6) Null insertion.	Allowed.	Allowed.	For empty TreeSet, 1st element null insertion is possible in all other cases get NullPtrExcep.

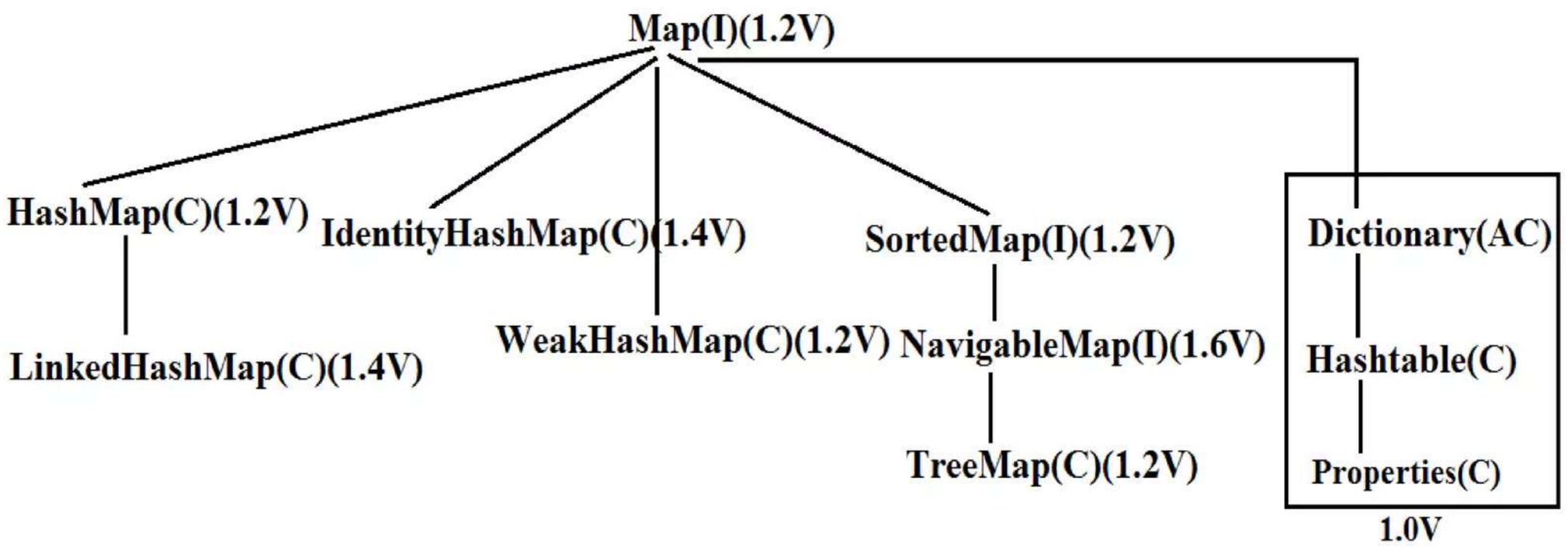
QUESTIONS →
QQ:1. Set allows at most one null element
a.True
b.False

QQ:2. Which implementation of Set should we use if we want the iterator to retrieve the objects in the order we have inserted?
a.TreeSet
b.HashSet
c.LinkedHashSet

QQ:3. If we need to store user defined objects in a TreeSet, which interface should the corresponding class implement?
Set
TreeSet
Comparable Interface
HashSet

MAP →

- ✓ Represent a group of objects as "key-value" pair (= one entry).
- ✓ Both key and value are objects only.
- ✓ Duplicate keys - NOT Allowed BUT *values* can be *Duplicated.*



NOTE: Map- Not Child Interface of Collection => Can't apply Collection Interface mthds.

QQ. TreeSetmap = new TreeSet();
map.add("one");
map.add("two");
map.add("three");
map.add("one");
map.add("four");
Iterator it = map.iterator();
while (it.hasNext()) {

```
    System.out.print( it.next() + " " );
}
```
a.Compilation fails
b.four three two one
c.one two three four
d.four one three two

QQ:
```
public static void before() {
    Set set= new TreeSet();
    set.add("2");
    set.add(3);
    set.add("1");
    Iterator it = set.iterator();
    while (it.hasNext())
    System.out.print(it.next() + " ");
    }
```
Which of the following statements are true?
a.The before() method will print 1 2
b.The before() method willprint 1 2 3
c.The before() method will not compile. (correct)

GENERICS →

Mechanism by which single piece of code can manipulate many different data types without explicitly having a separate entity for each datatype.

- ✓ Main objective to provide Type-Safety [Type of object/elements DS holding] and to resolve Type-Casting problems.
- ✓ Generic class can have multiple type parameters.
- ✓ Collections are NOT Type-Safe i.e., can't provide any guarantee for the type of elements present inside collection.
- ✓ Eg.: if our programming requirement is to hold only string type of objects it is never recommended to go for ArrayList.

Eg.:

```
ArrayList l=new ArrayList();
l.add("vijaya");
l.add("bhaskara");
l.add(new Integer(10));
.
.
.
String name1=(String)l.get(0);
String name2=(String)l.get(1);
String name3=(String)l.get(2);(invalid)
```
R.E → Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

##. *Before Generics* -

```
List myIntegerList = new LinkedList();
myIntegerList.add(new Integer(0));
Integer x = (Integer) myIntegerList.iterator().next(); // if not properly typecasted
                                                       // will throw runtime exception.
```

##. *After Generics* -

```
List<Integer> myIntegerList = new LinkedList<Integer>();
myIntegerList.add(new Integer(0));
Integer x = myIntegerList.iterator().next();        // No need for typecasting here
```

AutoBoxing with Collections →

Boxing conversion converts primitive values to objects of corresponding wrapper type

```
int i=11;
Integer iReference=newInteger(i);
iReference=i;
```

Unboxing conversion converts objects of wrapper types to values of corresponding primitive types.

```
int j=iReference.intValue();
j=iReference;
```

QQ. Which of the following declarations are allowed?

1. ArrayList<String> I1=new ArrayList<String>(); //(valid)
2. ArrayList<?> I2=new ArrayList<String>(); //(valid)
3. ArrayList<?> I3=new ArrayList<Integer>(); //(valid)
4. ArrayList<? extends Number> I4=new ArrayList<Integer>(); //(valid)
5. ArrayList<? extends Number> I5=new ArrayList<String>(); //(invalid) CTE
6. ArrayList<?> I6=new ArrayList<? extends Number>(); // CTE
7. ArrayList<?> I7=new ArrayList<?>(); // Err – class / Interface without bound.

Remember:

- Generics concept is applicable only at compile time, at runtime there is no such type of concept. Hence the following declarations are equal.

```
ArrayList I=new ArrayList<String>();
ArrayList I=new ArrayList<Integer>();     // All are equal at runtime.
ArrayList I=new ArrayList();
```

- Below 2 declarations are equal -

```
ArrayList<String> l1=new ArrayList();
ArrayList<String> l2=new ArrayList<String>();
```

>> For above ArrayList objects we can add only String type of objects.

```
l1.add("A");//valid
l1.add(10); //invalid
```

Difference between Serialization & Externalization :

Serialization	Externalization
It is meant for default Serialization	It is meant for Customized Serialization
Here every thing takes care by JVM and programmer doesn't have any control	Here every thing takes care by programmer and JVM doesn't have any control.
Here total object will be saved always and it is not possible to save part of the object.	Here based on our requirement we can save either total object or part of the object.
Serialization is the best choice if we want to save total object to the file.	Externalization is the best choice if we want to save part of the object.
relatively performence is low	relatively performence is high
Serializable interface doesn't contain any method , and it is marker interface.	Externalizable interface contains 2 methods 1.writeExternal() 2. readExternal() NOT a marker interface.
Serializable class not required to contains public no-arg constructor.	Compulsory to contains public no-arg constructor else will get RuntimeException saying "InvalidClassException"
Transient keyword play role in serialization	No role Transient keyword.

I/O Streams →

- ✓ Java programs perform I/O through **streams**.
- ✓ **A stream is**– an abstraction that either produces or consumes information.
  – linked to a physical device by the Java I/O system.
- ✓ All streams behave similarly, even if the actual physical devices to which they are linked differ.
- ✓ Java's stream classes are defined in the java.iopackage.

➢ Java defines 2 types of streams:

1. Byte streams - Provide convenient mean for handling input & output of bytes.
   - Used for reading or writing binary data.
2. Character streams - For handling input & output of characters.
   - Use Unicode and therefore can be internationalized.

Byte Stream classes →

BufferedInputStream
BufferedOutputStrea
m

To read & write data into buffer

FileInputStream
FIleOutputStream

ObjectInputStream
ObjectOutputStrea
m

To read & write object into secondary device (serialization)

Character Stream classes →

BufferedReader
BufferedWriter

To read & write data into buffer

FileReader
FIleWriter

To read & write data into file

InputStreamReade
r
OutputStreamWrit
er

Bridge from character stream to byte stream

The Predefined Streams →

- ✓ System class of java.lang package contains 3 predefined stream variables,in,outanderr.
- ✓ These variables declared as public and static within System:
  1. **System.out** - Refers to standard output stream which is the console.
  2. **System.in** - standard input, which is the keyboard by default.
  3. **System.err** - standard error stream, which also is the console by default.
- ➢ *The standard output should be used for regular program outputs while standard error should be used for error messages.*

System Properties →

Key	Description of Associated Value
java.version	Java Runtime Environment version
java.home	Java installation directory
java.class.path	Java class path
os.name	Operating system name
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

QQ. Java input output classes are available in _____package
a.java.lang
b.java.io
c.java.util

QQ. Can data flow through a given stream in both directions?
a.Yes
b.No

INTERVIEW QUESTIONS →

1	What is the difference between ArrayList and Vector?
2	What is the difference between ArrayList and LinkedList?
3	What is the difference between Iterator and ListIterator?
4	What is the difference between Iterator and Enumeration?
5	What is the difference between List and Set?
6	What is the difference between HashSet and TreeSet?
7	What is the difference between Set and Map?
8	What is the difference between HashSet and HashMap?
9	What is the difference between HashMap and TreeMap?
10	What is the difference between HashMap and Hashtable?
11	What is the difference between Collection and Collections?
12	What is the difference between Comparable and Comparator?
13	What is the advantage of Properties file?
14	What are the classes implementing List interface?
15	What are Collection related features in Java 8?
16	What is Java Collections Framework? List out some benefits of Collections framework?

17	What is the benefit of Generics in Collections Framework?
18	What are the basic interfaces of Java Collections Framework?
19	Why Collection doesn't extend Cloneable and Serializable interfaces?
20	Why Map interface doesn't extend Collection interface?
21	How can Arraylist be synchronized without using Vector?
22	When to use ArrayList or LinkedList ?
23	What are advantages of iterating a collection using iterator?
24	Why can't we create generic array? or write code as List<Integer>[] array = new ArrayList<Integer>[10];
25	Why can't we write code as List<Number> numbers = new ArrayList<Integer>();?
26	What is difference between Comparable and Comparator interface?
27	What is Collections Class?
28	What is Comparable and Comparator interface?
29	What are similarities and difference between ArrayList and Vector?
30	How to decide between HashMap and TreeMap?
31	What are different Collection views provided by Map interface?
32	What is the importance of hashCode() and equals() methods?

Based on Thread Control & Priorities →

1	What is Thread in Java?
2	Difference between Thread and Process in Java?
3	How do you implement Thread in Java?
4	When to use Runnable vs Thread in Java?
5	Difference between start() and run() method of Thread class?
6	How to stop thread in Java?
7	Explain the different priorities of Threads.
8	What if we call run() method directly instead of start() method?
9	What is the primary drawback of using synchronized methods?
10	Is it possible to start a thread twice?
11	What are the different states of a thread?
12	What are the different ways in which thread can enter the waiting stage?