Problems with Array:

1. Arrays are fixed in size and if we know the correct size of array before declaring then it's fine to go with array but in real life applications it happens rarely that we would know size in advance.
2. Second limitation of Array is, it can hold elements of same data type.
3. There is no underlying data structure for array because of that readymade methods are not available. In order to implement methods, we will have to define methods explicitly. Which increase complexity of code.

To overcome above problems of Array, Java provided Collection framework.

1. Collections are growable in nature, that is based on our requirement, we can increase or decrease the size.
2. Collections can hold both homogeneous and heterogeneous elements.
3. Every collection class is implemented based on some standard data structure hence for every requirement readymade method support is available. As a programmer we are responsible to use those methods not responsible to implement those methods.

Differences between Arrays and Collections

Arrays	Collection
Arrays are fixed in size that is once we create an array, we can't increase or decrease the size based on our requirement.	Collections are growable in nature that is based on our requirement, we can increase or decrease the size.
With respect to memory Arrays are not recommended to use.	With respect to memory collections are recommended to use.
With respect to performance, Arrays are recommended to use. **Example:** New element can be added easily if there will be space in array.	With respect to performance, Collections are not recommended to use. **Example:** If we are trying to add new object in collection and its initial capacity will be filled up then it will create new collection with new capacity and copy all objects from previous collection and add in newly created collection. Previous collection object will be referred to this newly created collection and previous will be handover to garbage collection for clean, this process will be time consuming and reduce performance.
Arrays can hold only homogeneous data type elements.	Collections can hold homogeneous and heterogeneous elements.
There is no underlying data structure for Array and hence readymade methods support is not available. For every requirement we have to write code explicitly which increases complexity of programming.	Every collection class is implemented based on some standard data structure and hence for every requirement readymade method support is available. We don't have to implement methods, we can use those method directly.

Arrays can hold both primitives and objects	Collections can hold only object type but not primitives.

Collections:

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

Collection Framework:

It contains several classes and interfaces which can be used to represent a group of individual objects as single entity.

1. Collection (Interface):

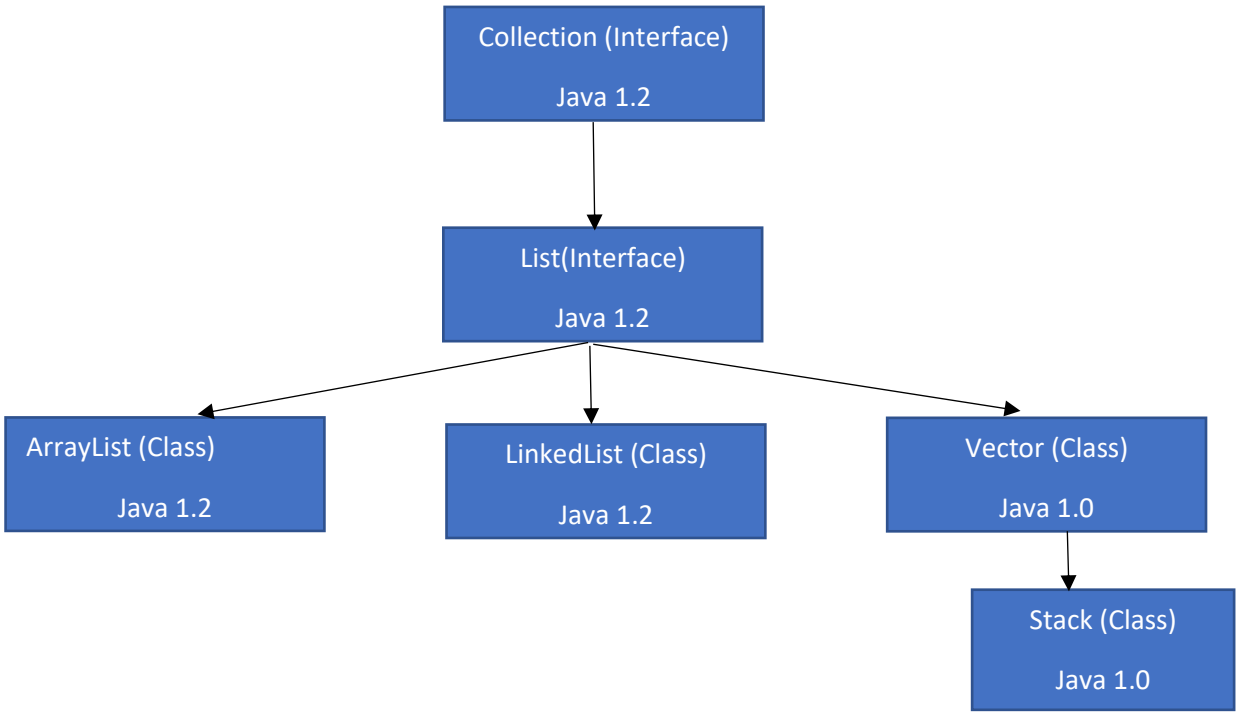
1. If we want to represent group of individual objects as a single entity then we should go for collection.
2. Collection interface defines most common methods which are applicable for any collection object.
3. In general, collection interface considered as root interface of collection framework.
4. There is no concrete class which implements collection interface directly.

Difference between Collection and Collections.

Collection is an interface, if we want to represent group of individual object as single entity then should go for collection.

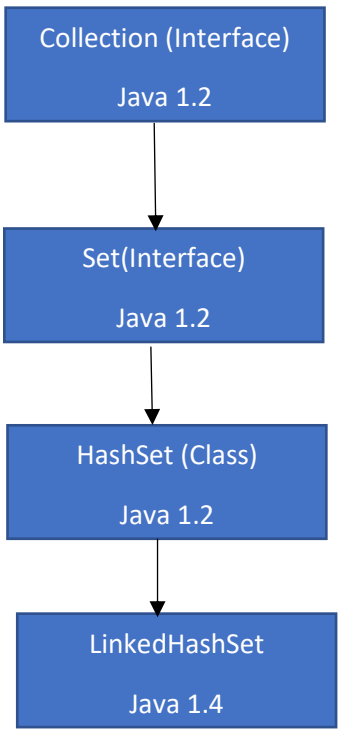
Collections is an utility class present in **java.util** package to define several utility methods for collection objects. Example: sort (), search () etc.

2. List (Interface): It is child interface of collection, if we want to represent a group of individual objects as a single entity where **duplicates are allowed** and **insertion order must be preserved** then we should go for List interface.



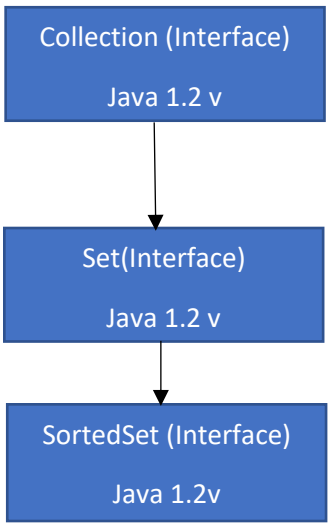
Note: In 1.2 version Vector and stack classes are reengineered to implement list interface.

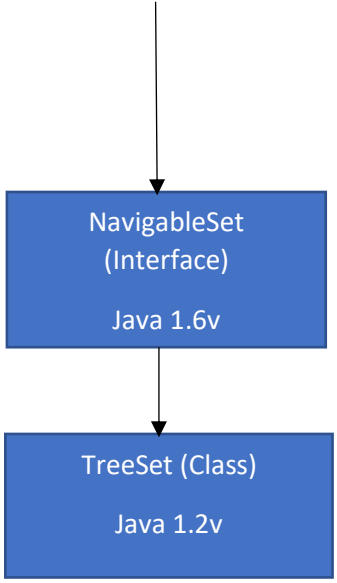
3. Set (Interface): It is child interface of collection. If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order not required then we should go for Set interface.



4. SortedSet (Interface): It is child interface of Set interface. If we want to represent a group of individual objects as a single entity where duplicates are not allowed and all object should be inserted according to some sorting order then we should go for SortedSet.

5. NavigableSet (Interface): It is child interface of Sorted set. It contains several methods for navigation purposes.



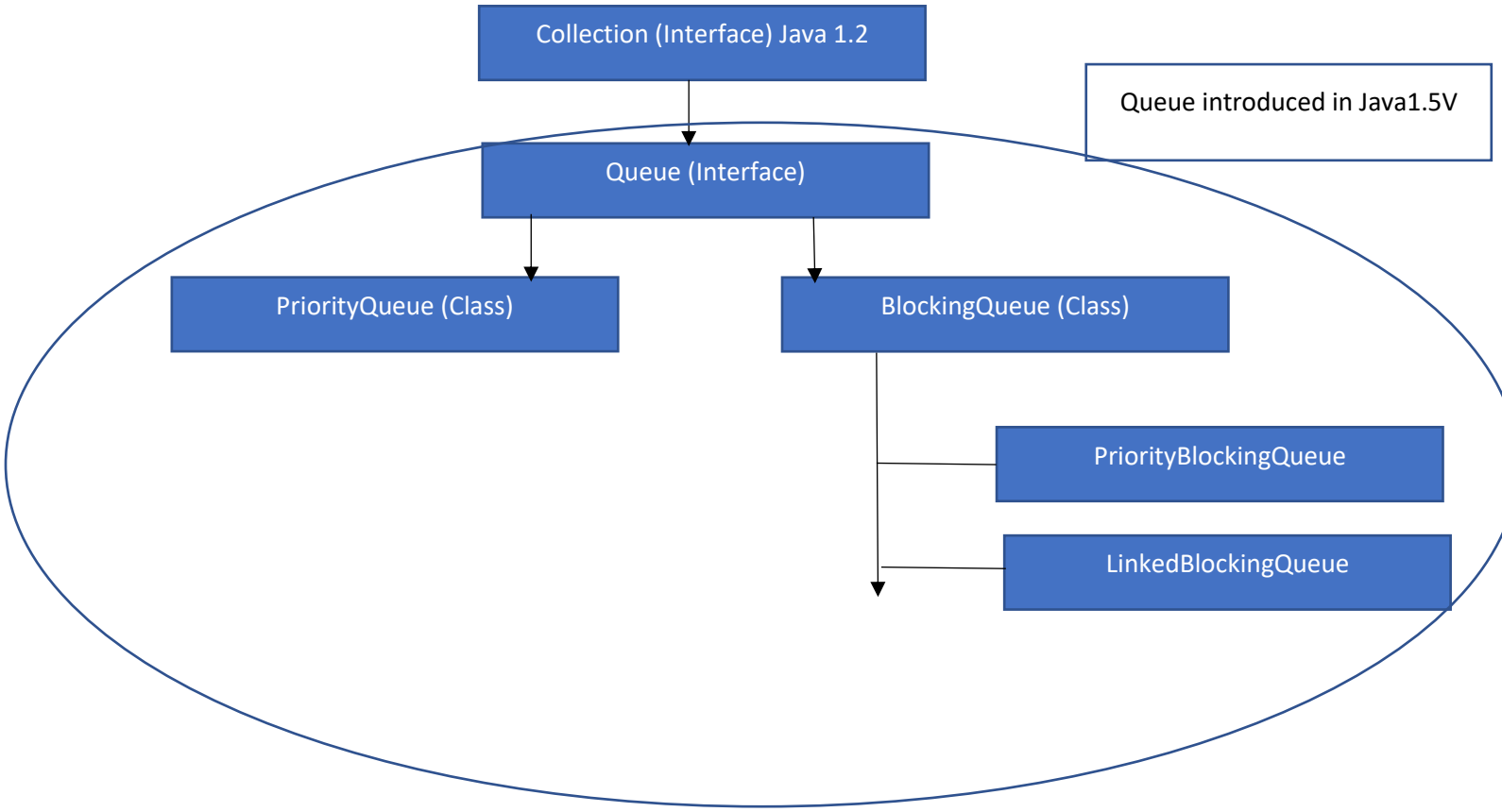


Differences Between List and Set:

List	Set
Duplicates are allowed	Duplicates not allowed
Insertion order is preserved	Insertion order not preserved.

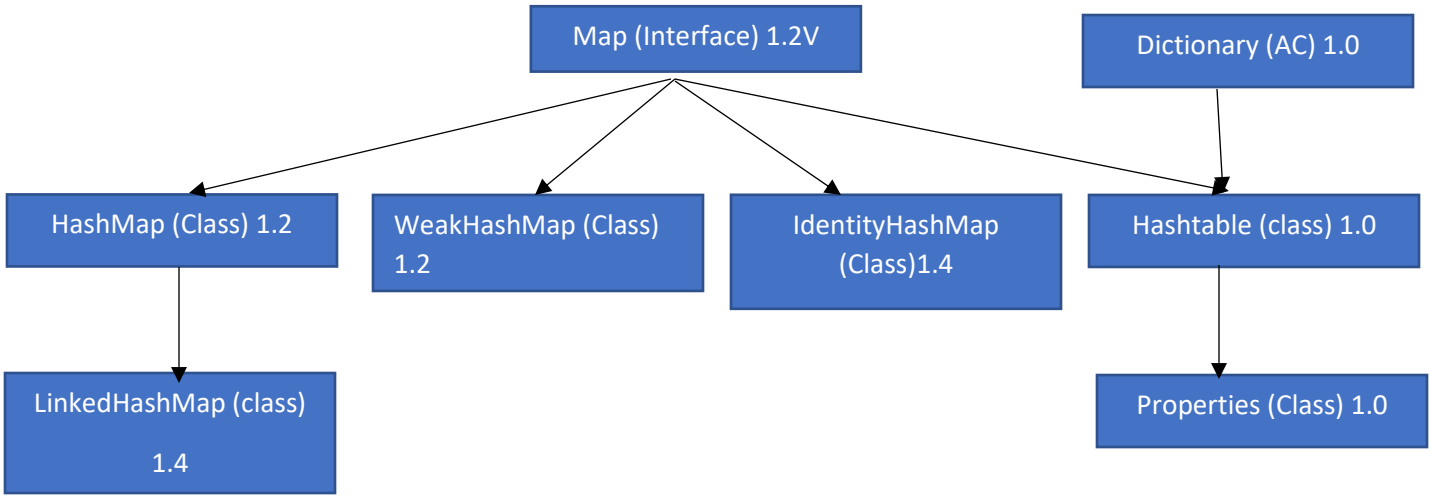
6. Queue (Interface): It is child interface of collection interface. If we want to represent a group of individual objects **prior to processing** then we should go for queue. Usually Queue follows FIFO order but based on our requirement we can implement our own priority order also.

Example: Before sending an email, all mail Ids we have to store in some data structure. In which order we added mail Ids in the same order only mail should be delivered. For this requirement Queue is best choice.



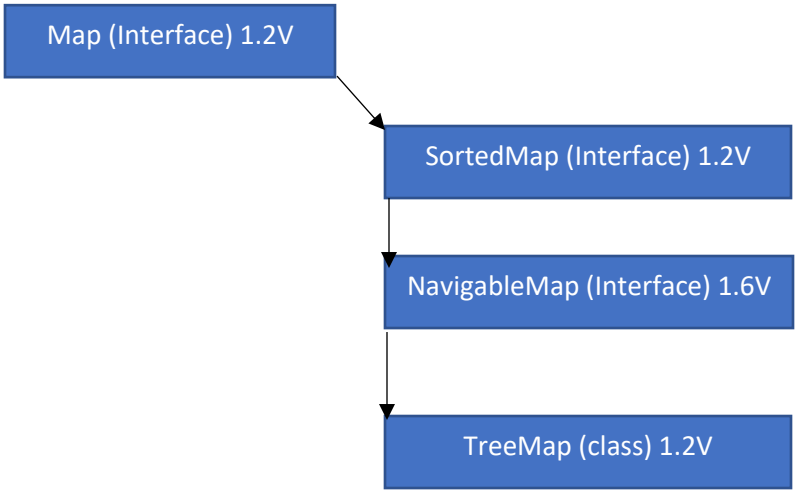
7. Map (Interface): Map is not child interface of Collection. If we want to represent a group of objects as Key Value pair then we should go for Map.

Key	Values
Roll No.	**Name**
101	John
102	James
103	Kim



8. SortedMap (Interface): It is child interface of Map. If we want to represent a group of key value pair according to some **sorting order of keys** then we should use SortedMap interface. In sorted map the sorting should be based on keys not based on values.

9. NavigableMap (Interface): It is child interface of SortedMap. It defines several methods for Navigation purposes.



Collection Interface

Common Methods / Generalized Methods present in Collection Interface

1. **boolean add(Object o);**
2. **boolean addAll(Collection c);**
3. **boolean remove(Object o);**
4. **boolean removeAll(Object o);**
5. **boolean retailAll(Collection c) ; → remove all object except present in c collection**
6. **void clear(); → remove all objects in collection**
7. **boolean contains(Object o) ;**
8. **boolean containsAll(Collection c);**
9. **boolean isEmpty();**
10. **int size();**
11. **Object[] toArray();**
12. **Iterator iterator();**

Note: There is no concrete class which implements collection interface directly.

List Interface

We can preserve insertion order using index and we can differentiate duplicate objects by using index hence index will play very important role in List.

List interface defines the following 8 specific methods.

1. **void add(int index, Object o);**
2. **boolean addAll(int index, Collection c);**
3. **Object get(int index);**
4. **Object remove(int index);**
5. **Object set(int index, Object new) → use to replace existing object at specified index.**
6. **int indexOf(Object o) → returns index of first occurrence of Object o**
7. **int lastIndexOf(Object o)→ return index of occurrence of Object o**
8. **ListIterator listIterator(); to iterate through list Collection.**

ArrayList

There are some key points to note about ArrayList

1. Underlying data structure is resizable array or growable array.
2. Duplicate objects are allowed.
3. Insertion order is preserved.
4. Heterogeneous objects are allowed
5. Null insertion is possible.

Constructors of ArrayList

1. **ArrayList list = new ArrayList ();**

   default constructor with default initial capacity (10). Once ArrayList reaches to its highest capacity that is 10 and I want to add 11th object then new ArrayList object will be created with new capacity and all previous objects will be copied to new ArrayList object and reference will be given to newly created list and previous will be deleted. New ArrayList capacity will be calculated using formula

   New Capacity = (current capacity*3/2)+1

2. **ArrayList list = new ArrayList(int initialCapacity);**

   In case if we want to define customized initial capacity of ArrayList then we can use this constructor and can define initial capacity at the time of creating object of ArrayList.

3. **ArrayList list = new ArrayList(Collection c);**

   Creates an equivalent ArrayList object for the given collection. It means any equivalent collection will be converted in type of ArrayList.

Example for ArrayList:

```
package com.citiustech;
import java.util.ArrayList;
public class ArrayListEx {
	public static void main(String[] args) {
		ArrayList list = new ArrayList();
		list.add("A");
		list.add(10);
		list.add(20);
		list.add(null);
		System.out.println(list);
		list.remove(2);
		System.out.println(list);
		list.add(2, "Bhushan");
		list.add("John");
		System.out.println(list);

	}
}
```

Important Note: Usually we can use collections to hold and transfer objects from one location to another location, to provide support for this requirement every collection class by default implements Serializable and Cloneable interface. ArrayList and Vector implements **RandomAccess** interface. Because

from ArrayList and Vector we are able to access any element randomly from any index. Therefor if our frequent operation is retrieval operation then Arraylist or Vector are best choices. RandomAccess interface present in java.util package and it doesn't contain any method therefor it is called as Marker interface. Where required ability will be provided automatically by JVM. To demonstrate above statements try following code.

```
package com.citiustech;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.RandomAccess;

public class ArrayListEx {

	public static void main(String[] args) {
		ArrayList list = new ArrayList();
		System.out.println(list instanceof Serializable);
		System.out.println(list instanceof Cloneable);
		System.out.println(list instanceof RandomAccess);
	}

}
```

Important point:
Array List is best choice if our frequent operation is retrieval of object because ArrayList implements RandomAccess interface. ArrayList is worst choice if our frequent operation is insertion or deletion in the middle.

Differences Between ArrayList and Vector

ArrayList	Vector
Every method present in ArrayList is non-Synchronized.	Every method present in Vector are Synchronized.
It is not thread safe	It is thread safe
Relatively performance is better than Vector	Relatively performance is low
ArrayList introduced in 1.2 so it's not legacy	Vector introduced in 1.0 so called as Legacy.

Question: How to get Synchronized version of ArrayList Object?
Ans: By default ArrayList is non-Synchronized but we can get Synchronized ArrayList object by using synchronizedList() method of Collections class.

Method:

public static List synchronizedList(List l)

Example:
```
ArrayList list = new ArrayList();
List l1 = Collections. synchronizedList(list);
```

Similarly, we can get Synchronized version of Set and Map objects by using following methods of Collections class.

- **public static Set synchronizedSet(Set s)**
- **public static List synchronizedMap(Map m)**

LinkedList

- Underlying data structure is doubly link list.
- Insertion order is preserved.
- Duplicates Objects are allowed
- Heterogeneous objects are allowed.
- Null insertion is possible.
- LinkedList implements Serializable and Cloneable interfaces but not RandomAccess
- LinkedList is best choice if our frequent operation is insertion or deletion in middle
- LinkedList is worst choice if our frequent operation is retrieval.

Constructors of LinkedList

1. **LinkedList list = new LinkedList ();**

   Creates an empty LinkedList object. There is no initial capacity because it is not going to save in contiguous way.

2. **LinkedList list = new LinkedList (Collection c);**
   Creates an equivalent LinkedList object for given collection.

LinkedList Class specific methods:

Usually we can use LinkedList to develop Stacks and Queues to provide support for this requirement LinkedList defines the following specific methods.

- **void addFirst(Object 0)**
- **void addLast(Object 0)**
- **Object getFirst()**
- **Object getLast()**
- **Object removeFirst()**
- **Object removeLast()**

Example for LinkedList:

```
package com.citiustech;
import java.util.LinkedList;
public class LinkedListEx {
	public static void main(String[] args) {
		LinkedList list = new LinkedList();
```

```
                list.add("John");
                list.add(1, "James");
                System.out.println(list);
                list.add(0,"Simon");
                System.out.println(list);
                list.addFirst("Kim");;
                list.addLast(null);
                System.out.println(list);
                list.removeFirst();
                list.removeLast();
                System.out.println(list);
                }
}
```

Vector

There are some key points to note about Vector

6. Underlying data structure is resizable array or growable array.
7. Duplicate objects are allowed.
8. Insertion order is preserved.
9. Heterogeneous objects are allowed
10. Null insertion is possible.
11. It is Thread safe

Constructors of ArrayList

1. **Vector vector = new Vector();**

   default constructor with default initial capacity (10). Once Vector reaches to its highest capacity that is 10 and I want to add 11th object then new Vector object will be created with new capacity and all previous objects will be copied to new Vector object and reference will be given to newly created vector and previous will be deleted. New Vector capacity will be calculated using formula

   New Capacity = (current capacity*2);

2. **Vector vector = new Vector(int initialCapacity);**

   In case if we want to define customized initial capacity of Vector then we can use this constructor and can define initial capacity at the time of creating object of Vector.

3. **Vector vector = new Vector(int initialCapacity, int incrementalCapacity);**

   Create Vector object with customized initial capacity and incrementation

Example:

Vector vector = new Vector(100,5);

So, once vector will reaches to its highest capacity then new vector Object will be created with size 100+5=105 capacity.

4. **Vector vector = new Vector(Collection c);**
   Creates an equivalent Vector object for given collection.

Vector Specific Method:

1. **add(Object 0); – method from Collection**
2. **add(int index, Object o); -- method from List**
3. **addElement(Object 0); ---Vector specific method.**
4. **remove(Object o); – method from Collection**
5. **removeElement(Object o); ---Vector specific method.**
6. **remove(int index) --- method from List**
7. **removeElement(int index); -- Vector specific method.**
8. **clear(); – method from Collection**
9. **removeAllElement();-- Vector specific method.**
10. **Object get(int index); ---- method from List**
11. **Object elementAt(int index); -- Vector specific method.**
12. **Object firstElement(); -- Vector specific method.**
13. **Object lastElement(); -- Vector specific method.**
14. **Enumeration elements(); -- Vector specific method. Use to retrieve elements one by one.**

Example for understanding Vector and it's Capacity.

```
package com.citiustech;

import java.util.Vector;

public class VectorEx {
	public static void main(String[] args) {
		Vector vector = new Vector();
		System.out.println(vector.capacity());
		for (int i = 1; i <=10; i++) {
			vector.add(i);
		}
		System.out.println(vector.capacity());
		vector.add("A");
		System.out.println(vector.capacity());
	}
}
```

Stack

Stack Specific Methods:

1. **Object push (Object o) -- to insert an object into stack**
2. **Object pop () – to remove and return top of stack**
3. **Object peek() – to return top of the stack without removal**
4. **boolean empty() – return true if stack is empty.**
5. **int search(Object o) – return offset if the element is available otherwise returns -1. (Offset mean searching from tops)**

Example:

```
package com.citiustech;
import java.util.Stack;
public class StackEx {
public static void main(String[] args) {
      Stack stack = new Stack();
      stack.push("A");
      stack.add(0, "Bhushan");
      System.out.println(stack);
      System.out.println(stack.search("Bhushan"));
      System.out.println(stack.get(0));
}
}
```

Cursors:

So far, we have seen some implemented classes of List interface and we created object for them. We can also get objects in collection object one by one. In order to achieve this, we can use cursors.

There are three cursors in Java

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

Enumeration: (1.0)

We can use Enumeration to get objects one by one from legacy collection object. We can create enumeration object by using element method of Vector class.

public Enumeration elements();

Example: Enumeration e= vector.elements();

Methods of Enumeration:

1. **public boolean hasMoreElement();**
2. **public Object nextElement();**

Example:

```
package com.citiustech;

import java.util.Enumeration;
import java.util.Vector;

public class EnumeratorEx {

     public static void main(String[] args) {
          Vector vector = new Vector();
          for (int i = 0; i < vector.capacity();
i++) {
               vector.add(i);

          }
          System.out.println(vector);
          Enumeration enumeration =
vector.elements();
          while(enumeration.hasMoreElements()) {
               Integer
number=(Integer)enumeration.nextElement();
               if(number%2==0) {
               System.out.println(number);
               }
               else {
                    System.out.println(number+"
Removed");
                    vector.remove(number);
                    System.out.println(vector);
               }
          }

     }
```

```
}
```

Limitation of Enumeration cursor.

1. We can apply Enumeration only for legacy classes and it is not a universal cursor.
2. By using Enumeration, we can get only read access and we can't perform remove operation. To overcome above limitation, we can use Iterator interface.
3. Using Enumeration we can move only in forward direction.

Iterator:(1.2)

1. We can use Iterator for any collection object therefor it is called as universal cursor.
2. By Using Iterator, we can perform both read and remove operations

We can create Iterator object by using iterator() method of Collection interface.

public Iterator iterator()

Example:

Iterator itr = c.iterator();

Methods in Iterator:

1. **public boolean hasNext()**
2. **public Object next()**
3. **public void remove()**

Example:

```
package com.citiustech;

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

public class IteratorEx {
	public static void main(String[] args) {
		ArrayList list = new ArrayList();
		for (int i = 0; i < 10; i++) {
			list.add(i);
		}
			System.out.println(list);
			Iterator itr = list.iterator();
			while(itr.hasNext()) {
				Integer number=(Integer)itr.next();
				if(number%2==0) {
				System.out.println(number);
				}
				else {
					System.out.println(number+" Removed");
					itr.remove();
```

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

        }
```

Limitation of Iterator:

1. Using Iterator, we can move only in forward direction like Enumeration. We can't move in backward direction. Therefor is called as single directional cursor.
2. Using Iterator, we can perform only read and remove operations. We can't perform replacement and addition of new objects in collection.

To overcome above limitations, we should go for **ListIterator.**

ListIterator:(1.2)

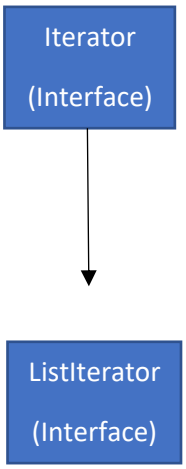
1. Using ListIterator, we can move either to forward direction or to the backward direction therefore it is called as Bidirectional cursor.
2. Using ListIterator, we can perform replacement and addition of new object in addition to read and remove operations.

We can create Iterator object by using listIterator() method of List interface.

public ListIterator ListIterator()

Example:

ListIterator litr = l.listIterator();



Methods of ListIterator:

ListIterator is child interface of Iterator therefor all methods present in Iterator by default available to ListIterator.

ListIterator defines following 9 methods.

For forward direction:

1. **public boolean hasNext()**
2. **public Object next()**
3. **public int nextIndex()**

For backward direction:

1. **public boolean hasPrevious()**
2. **public Object previous()**
3. **public int previousIndex()**

Extra method for add, replace, remove

1. **public void remove()**
2. **public void add(Object o);**
3. **public void set(Object o);**

This are 9 methods are for ListIterator.

Example:

```
package com.citiustech;

import java.util.LinkedList;
import java.util.ListIterator;

public class ListIteratorEx {

	public static void main(String[] args) {
		LinkedList list = new LinkedList();
		list.add("John");
		list.add("James");
		list.add("Kevin");
		list.add("Tom");
		System.out.println(list);
		ListIterator litr= list.listIterator();
		while(litr.hasNext()) {
			String names = (String)litr.next();
			if(names.equals("John")) {
				litr.remove();
			}
			else if(names.equals("Tom")) {
				litr.add("Zanillia");
			}
			else if(names.equals("Kevin")) {
				litr.set("Vinod");
			}
			System.out.println(list);
		}
	}

}
```

Example 2:

```
package com.citiustech;

import java.util.LinkedList;
import java.util.ListIterator;

public class ListIteratorEx {

	public static void main(String[] args) {
		LinkedList list = new LinkedList();
		list.add("John");
		list.add("James");
		list.add("Kevin");
		list.add("Tom");
		System.out.println(list);
		ListIterator litr= list.listIterator();

		System.out.println(litr.next());
		System.out.println(litr.nextIndex());
		System.out.println(litr.next());
		System.out.println(litr.previous());
		System.out.println(litr.previousIndex());

	}

}
```

Note:

ListIterator can use only for List objects.

Question: If all cursors are interfaces then how does we are able to create objects for them?

Ans: We are creating objects for respective interface by using methods present in implemented class. You can understand with given below example:

```
public static void main(String[] args) {
		Vector vector = new Vector();
		Enumeration enumeration = vector.elements();
		Iterator itr = vector.iterator();
		ListIterator litr = vector.listIterator();
		System.out.println(enumeration.getClass().getName());
		System.out.println(itr.getClass().getName());
		System.out.println(litr.getClass().getName());
	}
```

Output:(Red color text is description of output.)

```
java.util.Vector$1 → anonymous class implementation
java.util.Vector$Itr → Itr class to implement Iterator interface
java.util.Vector$ListItr → ListItr class to implement ListIterator interface.
```

Set Interface

Set is child interface of collection, if we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order doesn't matter then we can use Set interface.

Set interface doesn't contain any new method. We have to use only collection interface 12 methods.

HashSet

Key points for HashSet.

1. The underlying data structure is Hashtable.
2. Duplicate objects are not allowed
3. Insertion order is not preserved because insertion will be done base on Hashcode of object
4. Null insertion is possible but only once
5. Heterogeneous objects are allowed.
6. Implements Serializable, Cloneable interface but not RandomAccess interface.
7. HashSet is best choice if our frequent operation is searching.

Note: In HashSet, duplicates are not allowed. If we are trying to insert duplicates then we won't get any compile or runtime error and add() method will return **false**.

Example:

```
HashSet set = new HashSet();

System.out.println(set.add("John"));  //print true

System.out.println(set.add("John"));  // print false
```

Constructors of HashSet:

Note:- For all classes which is having Hash, all of them having same constructors.

1. **HashSet set = new HashSet ();**

   Creates an empty HashSet object with default initial capacity 16 and default fill ratio or load factor is 0.75.

2. **HashSet set = new HashSet (int initialCapacity);**
   Creates an empty HashSet object with initial capacity of provided initial capacity value and default fill ratio or load factor is 0.75.

3. **HashSet set = new HashSet (int initialCapacity, float fillRatio);**
   Creates an empty HashSet object with initial capacity of provided initial capacity value and customized fill ratio/load factor specified in constructor.

4. **HashSet set = new HashSet (Collection c);**
   Creates an equivalent HashSet for given collection. This constructor meant for interconversion between collection objects.

Question: What is fill ration or load factor?

Ans: After filling how much ration a new HashSet object will be created, this ratio is called fill ratio or load factor.

Example:

Fill ratio 0.75 means after filling 75% ration a new HashSet object will be created.

Example:

```
package com.citiustech;

import java.util.HashSet;

public class HashSetEx {
	public static void main(String[] args) {
		HashSet set = new HashSet();
		set.add("John");
		set.add("James");
		set.add("Kevin");
		set.add(null);
		System.out.println(set.add("John"));
		System.out.println(set);
	}
}
```

LinkedHashSet

It is child class of HashSet, it is completely similar to HashSet including constructors and methods except following differences.

HashSet	LinkedHashSet
HashSet is using **Hashtable** as underlying data structure.	LinkedHashSet is using underlying data structure is **combination of LinkedList + Hashtable**
Insertion order is not preserved.	Insertion order is preserved
Introduced in Java 1.2	Introduced in Java 1.4

Example:

```
package com.citiustech;

import java.util.HashSet;

public class HashSetEx {
	public static void main(String[] args) {
		LinkedHashSet set = new LinkedHashSet();
		set.add("John");
		set.add("James");
		set.add("Kevin");
		set.add(null);
		System.out.println(set.add("John"));
		System.out.println(set);
	}
}
```

Note: In above code we have created object of LinkedHashSet therefor insertion order will be preserved.

SortedSet:(Interface)

SortedSet is child interface Set. If we want to represent group of individual objects according to some sorting order without duplicates then we can use SortedSet.

SortedSet defines the following 6 methods.

1. **Object first();**
   Return the first element of SortedSet
2. **Object last();**
   Return the last element of SortedSet.
3. **SortedSet headSet(Object obj)**

   Return SortedSet whose elements are less than obj

4. **SortedSet tailSet(Object obj)**
   Return SortedSet whose elements are >= obj
5. **SortedSet subset(Object obj1, Object obj2)**
   Return SortedSet whose elements >=obj1 and <obj2
6. **Comparator comparator()**
   Return comparator object that describes underlying sorting technique. If we are using default sorting order then we will get null value.

Example:

1. first() → 10
2. last() → 20
3. headSet(16) →[10,11,12,13,14,15]
4. tailSet(16) →[16,17,18,19,20]
5. subset(11,15) →[11,12,13,14]
6. comparator() → null

Consider we have set as given in blue box and If I will use methods of SortedSet then we will get output as above.

TreeSet:(Class)

1. **Underlying data structure is balanced tree**
2. **Duplicate objects are not allowed**
3. **Insertion order is not preserved**
4. **Heterogeneous object are not allowed otherwise we will get exception saying ClassCastException**
5. **null insertion is possible but only once.**
6. **Implements Serializable, Cloneable but not RandomAccess interface.**
7. **All object will be inserted based on some sorting order. It maybe default natural sorting order or customized sorting order.**

Constructors:

1. **TreeSet set = new TreeSet()**
   It creates empty TreeSet object where element will be inserted according to default natural sorting order.
2. **TreeSet set = new TreeSet (Comparator c);**
   It creates an empty TreeSet object where the elements will be inserted according to customized sorting order specified by comparator object.
3. **TreeSet set = new TreeSet (Collection c);**
4. **TreeSet set = new TreeSet (SortedSet set);**

Example:

```
package com.citiustech;

import java.util.TreeSet;

public class TreeSetEx {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		TreeSet set = new TreeSet();
		set.add("B");
		set.add("A");
		set.add("a");
		set.add("C");
		//set.add(null); //null pointer Exception
		//set.add(new Integer(10)); //ClassCastException
		System.out.println(set);
	}
}
```

Note: null acceptance.

1. In non empty TreeSet, we are trying to insert null then we will get NullPointerException.
2. For empty TreeSet as first element null is allowed but after inserting null if we will insert other then we will get RunTimeException saying NullPointerException.

Important Point: Till Java 1.6 null is allowed as first element to empty TreeSet but from java 1.7 null is not allowed even if it will be first element that is null is not allowed in TreeSet from Java 1.7 version.

Example:

```
package com.citiustech;

import java.util.TreeSet;

public class TreeSetEx {

	public static void main(String[] args) {
		TreeSet set = new TreeSet();
		set.add(new StringBuffer("A"));
		set.add(new StringBuffer("Z"));
		set.add(new StringBuffer("B"));
		set.add(new StringBuffer("L"));
		System.out.println(set);
	}
}
```

Note: If we are depending on default natural sorting order then objects must be **homogeneous and comparable** otherwise we will get RuntimeException **as ClassCastException.**

An object is said to be comparable if and only if corresponding class implements comparable interface. **String, and all wrapper classes are implements comparable interface. From Java 11 StringBuffer and StringBuilder also implementing Comparable Interface.**

Comparable Interface

1. It is present in java.lang package and
2. It contains only one method compareTo()

Definition of compareTo() method:

public int compareTo(Object obj)

1. If this method returns -ve means object1 come before object2
2. If this method returns +ve means object1 comes after object2
3. If this method returns 0 then both objects are equals.

Example to demonstrate above code:

```
package com.citiustech;

import java.util.TreeSet;

public class TreeSetEx {

	public static void main(String[] args) {
		System.out.println("A".compareTo("Z")); //-25
		System.out.println("Z".compareTo("K")); //15
		System.out.println("A".compareTo("A")); //0
		System.out.println("A".compareTo(null)); //NPE
	}
}
```

If we are depending on default natural sorting order then while adding object into TreeSet JVM will call compareTo() method

TreeSet set = new TreeSet();

set.add("K");

set.add("Z");

set.add("A");

set.add("A");

Sysout(set) // Output → [A K Z]

Obj1.compareTo(Obj2);

Note: Obj1 means object which is inserting and Obj2 means Object which is already inserted.

If default natural sorting order not available or if we are not satisfied with default natural sorting order then we can use customized sorting by using Comparator interface.

Comparable	Comparator
Use for default natural sorting order	Use for customized sorting order.

Comparator Interface

1. Comparator present in java.util package and it define two methods compare() and equals().

   Definition of methods:

   1. **public int compare(Object o1, Object o2)**
      a. -ve if o1 will come before o2
      b. +ve if o1 will come after o2
      c. 0 if both are equal
   2. **public boolean equals(Object obj)**

Example:

class MyClass implements Comparator {

public int compare(Object o1, Object o2){

}

// we are not implementing equals() of Comparator but we won't get any compile //time error why?

//Reason: because MyClass by default extending Object class and equals() method already defined in Object class therefor we don't have to implement equals() method

}

We will try to understand implementation of Comparator interface with one example.

```
package com.citiustech;
import java.util.Comparator;
import java.util.TreeSet;

public class TreeSetEx {

      public static void main(String[] args) {
            TreeSet set = new TreeSet(new
MyComparator());
            set.add(10);
            set.add(20);
            set.add(0);
            set.add(12);
            set.add(14);
            System.out.println(set);
```

```
        }

    }
    class MyComparator implements Comparator{

        @Override
        public int compare(Object o1, Object o2) {
            Integer n1=(Integer)o1;
            Integer n2=(Integer)o2;
            if(n1<n2)
                return +1;
            else if(n1>n2)
                return -1;
            else
                return 0;
        }

    }
```

Question: How does it work?

Ans: First we are going to create object of TreeSet with Comparator object in which we are going to implement our customized sorting order (Descending).

```
TreeSet set = new TreeSet(new MyComparator());
```

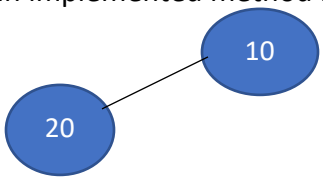
Then start adding object using add() method.

set.add(10);
so first and only one element has been added to TreeSet and there is nothing to compare. Now, we are adding second object
set.add(20);

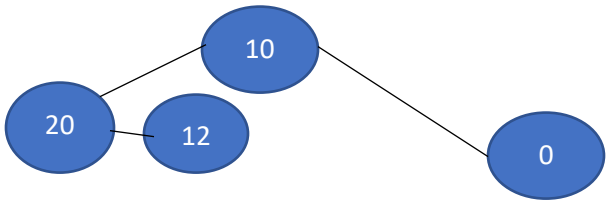
now, there is already an object is present so TreeSet will call implemented compare() method of MyComparator.

Compare(Object obj1, Object obj2); here Object1 is the object we are adding and object2 is that object which is already present there. So internally. It will call method as compare(20,10);

As per logic in implemented method it will return -1 so it will put-on left-hand side.



10 will be act as root element in this code. Now, set.add(0); then it will call method compare as compare(0,10); as per logic it will return +1 so 0 will be put on left side.



Next, set.add(12); then internally it will call method compare as compare(12,10); then as per logic it will return -ve so compare with object which is on left hand side and call method as compare(12,20) then it will return +1. So it will put on right hand side of 20 and so on.

At the end we will get output as

[20,14,12,10,0].

Conclusion: We are getting object sorted in descending order because we have used Comparator. If we would have used default constructor then it will be sorted with default sorted way that is ascending order.

Example 2: Possibilities of compare() method.

```
package com.citiustech;
import java.util.Comparator;
import java.util.TreeSet;

public class TreeSetEx {
	public static void main(String[] args) {
		TreeSet set = new TreeSet(new MyComparator());
		set.add(10);
		set.add(20);
		set.add(0);
		set.add(12);
		set.add(14);
		System.out.println(set);
	}
}
class MyComparator implements Comparator{

	@Override
	public int compare(Object o1, Object o2) {
		Integer n1=(Integer)o1;
```

```
            Integer n2=(Integer)o2;
            return n1.compareTo(n2); //default sorting order
            //return -n1.compareTo(n2);//descending order
            //return n2.compareTo(n1);// descending order
            //return -n2.compareTo(n1);//ascending order
            //return +1;//insertion order
            //return -1;//insertion order in reverse manure.
            //return 0;//only first object will be printed.
      }

}
```

Example 3: TreeSet with String objects

```
package com.citiustech;
import java.util.Comparator;
import java.util.TreeSet;

public class TreeSetEx {
      public static void main(String[] args) {
            TreeSet set = new TreeSet(new MyComparator());
            set.add("James");
            set.add("John");
            set.add("Kevin");
            set.add("Adam");
            set.add("Vivian");
            System.out.println(set);
      }
}
class MyComparator implements Comparator{

      @Override
      public int compare(Object o1, Object o2) {
            String n1=(String)o1;
            String n2=(String)o2;
            //return n1.compareTo(n2); //default sorting order
            //return -n1.compareTo(n2);//descending order
            //return n2.compareTo(n1);// descending order
            //return -n2.compareTo(n1);//ascending order
```

```
            //return +1;//insertion order
            //return -1;//insertion order in reverse manure.
            //return 0;//only first object will be printed.
      }

}
```

Important Note: In TreeSet Objects must be Homogeneous and comparable.

Example to sort String on the basis of length of String

```
package com.citiustech;
import java.util.Comparator;
import java.util.TreeSet;

public class TreeSetEx {
      public static void main(String[] args) {
            TreeSet set = new TreeSet(new MyComparator());
            set.add("abcdefghi");
            set.add("James");
            set.add("John");
            set.add("Kevin");
            set.add("Adam");
            set.add("Vivian");
            System.out.println(set);
      }
}
class MyComparator implements Comparator{

      @Override
      public int compare(Object o1, Object o2) {
            String n1=(String)o1;
            String n2=(String)o2;
            int length1=n1.length();
            int length2 = n2.length();
            if(length1<length2)
                  return -1;
            else if(length1>length2)
                  return +1;
            else
                  //return 0;//same length object won't be there
                  return n1.compareTo(n2);//all objects will be //there

      }}
```

Example for Comparator and Comparable with respect to User Defined class.

```
package com.citiustech;

import java.util.Comparator;
import java.util.TreeSet;

public class TreeSetWithProductClass {

	public static void main(String[] args) {
		Products p1=new Products(101, "Sofa", 10000d);
		Products p2=new Products(104, "Chair", 5000d);
		Products p3=new Products(106, "Phone", 8000d);
		Products p4=new Products(112, "Bottle", 100d);
		Products p5=new Products(10, "Mouse", 150d);
		TreeSet set = new TreeSet(new MyComparator1());
		set.add(p1);
		set.add(p2);
		set.add(p3);
		set.add(p4);
		set.add(p5);
		System.out.println(set);
	}

}
class Products implements Comparable{
	int productId;
	String productName;
	double productPrice;

	public Products(int productId, String productName, double productPrice)
{
		this.productId = productId;
		this.productName = productName;
		this.productPrice = productPrice;
	}
	@Override
	public String toString() {
		return "productId=" + productId + ", productName=" + productName
+ ", productPrice=" + productPrice
				+ "";
	}
	@Override
	public int compareTo(Object o) {
		int productId=this.productId;
		Products product = (Products)o;
		int productId2 = product.productId;
		if(productId<productId2)
			return -1;
		else if(productId>productId2)
			return 1;
```

```
            else
                  return 0;

      }

}
class MyComparator1 implements Comparator{

      @Override
      public int compare(Object o1, Object o2) {
            Products p1= (Products)o1;
            Products p2= (Products)o2;
            String productName=p1.productName;
            String productName2=p2.productName;
            return productName.compareTo(productName2);
      }

}
```

Map:

1. Map is not child interface of collection.
2. If we want to represent a group object as Key Value pairs then we can use Map.
3. Both keys and values are objects only
4. Duplicates keys are not allowed
5. Values can be duplicated.
6. Each key value pair is called **Entry** hence Map is considered as a collection of Entry objects.

Method of Map Interface:

1. Object put(Object key, Object value)

   map.put(10,"John"); → return null

   map.put(11,"James"); → return null

   map.put(10,"Kim"); → John object will return

in first two cases method will return because keys are different but in third case same key is there then "John" will replace by "Kim" so it will return old value.

2. **void putAll(Map map);**
3. **Object get(Object key);**
4. **Object remove(Object key);**
5. **boolean containsKey(Object key);**
6. **boolean containsValue(Object value);**
7. **boolean isEmpty();**
8. **int size()**

9. **void clear();**

Below 3 methods are called as Collection view of Map

10. **Set keyset()**
11. **Collection values()**
12. **Set entrySet()**

Entry Interface:

A Map is group of key value pairs and each key value is called an Entry. Hence Map is considered as a collection of Entry Objects. Without Map Object there won't be any Entry object therefor Entry Interface is defined inside Map interface.

interface Map {

interface Entry {

Object getKey()

Object getValue()

Object setValue(Object new)

} — Entry Specific methods and we can apply on Entry Object

}

HashMap:(Class)

1. Underlying Data structure is Hashtable.
2. Insertion order is not preserved and it is based on HashCode of Keys.
3. Duplicate keys are not allowed but values can be duplicated.
4. Heterogeneous objects are allowed for both and key and values.
5. Null allowed for key only once but for values multiple times
6. HashMap implements serializable and cloneable interfaces not RandomAccess.
7. HashMap is best choice if our frequent operation is Searching.

Constructors of HashMap:

1. **HashMap map = new HashMap();**

   It creates empty HashMap object with default initial capacity 16 and default fill ratio/load factor is 0.75.

2. **HashMap map = new HashMap(int initialCapacity);**
   We can customize initial capacity and default fill ratio is 0.75
3. **HashMap map = new HashMap(int initialCapacity, float fillRation);**
   We can customize initial capacity and default fill ratio can be customized
4. **HashMap map = new HashMap(Map map);**

Example for understanding HashMap:

```
package com.citiustech;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapEx {
	public static void main(String[] args) {
		HashMap map = new HashMap();
		map.put(101, "John");
		map.put(102, "Brian");
		map.put(103, "Kevin");
		map.put(104, "Jonathan");
		map.put(105, "James");
		System.out.println(map);
		System.out.println(map.put(103, "Kiara"));
		Set set = map.keySet();
		System.out.println(set);
		Collection collection= map.values();
		System.out.println(collection);
		Set mapEntrySet =map.entrySet();
		System.out.println(mapEntrySet);
		Iterator iterator = mapEntrySet.iterator();
		while(iterator.hasNext()) {
			Map.Entry m = (Map.Entry)iterator.next();
			System.out.println(m.getKey()+"--
"+m.getValue());
			if((Integer)m.getKey()==103) {
				m.setValue("Bhushan");
			}
		}
		System.out.println(map);
	}
}
```

Note: There is a Class similar to HashMap is Hashtable. It is similar to ArrayList and Vector. HashMap is not synchronized while Hashtable is synchronized. HashMap introduced in 1.2 but Hashtable in 1.0 and it is called as Legacy class. In Hashtable null are not allowed.

To make HashMap Synchronized we will have to use static method of Collections class that is synchronizedMap();

Queue:

Queue specific methods:

1. **boolean offer(Object obj)**

   to add object in Queue

2. **Object peek()**
   To return head element of the queue. If queue is empty then this method will return null

3. **Object element()**
   To return head element of the queue. If queue is empty then this method will throws exception "NoSuchElementException".

4. **Object poll()**
   To remove and return head element of the queue. If queue is empty then this method will return null

5. **Object remove()**
   To remove and return head element of the queue. If queue is empty then this method will throws exception "NoSuchElementException".

PriorityQueue

1. If you want to represent group of individual objects prior to processing according to some priority then we can use PriorityQueue.
2. Priority can be default natural sorting order or customized sorting order defined by Comparator.
3. Insertion order is not preserved because it is based on priority.
4. Duplicate objects are not allowed
5. If we are depending on default natural sorting order then object should be homogeneous and comparable otherwise we will get Runtime Exception **ClassCastException.**
6. If we are defining our own sorting by comparator then objects need not be homogeneous and comparable.
7. Null is not allowed even if it will be null element.

Example for PriorityQueue.

```
package com.citiustech;

import java.util.PriorityQueue;

public class QueueEx {
	public static void main(String[] args) {
		PriorityQueue queue = new PriorityQueue();
```

```
                System.out.println(queue.peek());
                //System.out.println(queue.element());
                for (int i = 0; i < 10; i++) {
                     queue.offer(i);
                }
                System.out.println(queue);
                System.out.println(queue.poll());
                System.out.println(queue);
                System.out.println(queue.poll());
          }
     }
```