Collection

Int a[]=new int[4];// Drawback: Array size is fixed

```
Object valuesOther[]=new Object[4];
valuesOther[0]="Kingshuk";
valuesOther[1]=2;
valuesOther[2]=4.77;
```

So, Collection is basically a dynamic array.

Notes:

Collection values=new Implementation ();

[Interface] [Implementation Class]

```
Collection values=new ArrayList<>();
values.add(6);
values.add(7);
```

In **Collection Interface** we do not have an option to insert in a particular position(as it does not maintain index).

And as it does not maintain index it cannot be sorted by **Collections.sort()**

Enhanced

List Interface that extends **Collection.**

But for List it has function add `list.add(index, object);`

We can sort by **Collections.sort()**

```
List list=new ArrayList<>();
```

- Maintain Insertion Order[Maintains index number that is why it have `add(index, object);`]
- But Duplicate Value Exist

```
Set set=new HashSet<>();
```

- Does not maintain Insertion order[we will randomly get the elements]
- No Duplicate value exist

```
TreeSet<>();
```

- When we print a TreeSet we get the elements in a Sorted Format

```
Map<Key,Value>
```

It's a key value relationship, corresponding to every key there is a value

List Traversing:

```
//Traversing of Arraylist
List list=new ArrayList<>();
//Conventional For Loop
for (int i=0;i<list.size();i++) {
    Object object = list.get(i);
}
//By while loop-- Iterator is a very old Technique
Iterator iterator=list.iterator();
while (iterator.hasNext()) {
    Object object = (Object) iterator.next();
}
//Enhanced For Loop
for(Object i:list) {
    System.out.println(i);
}
//Stream API using Lambda Expression-- Java 8 onwards
list.forEach(System.out::println);
```

Vector:

It is a dynamic array and it will increase size automatically.

```
Vector v=new Vector<>();
```

- By default, Vector Capacity: 10 ---- It multiplies once the size is full----20—40—80—160---
- Accessed by advance for loop.
- Vector is old—Even before Collection framework[Since JDK 1.0]

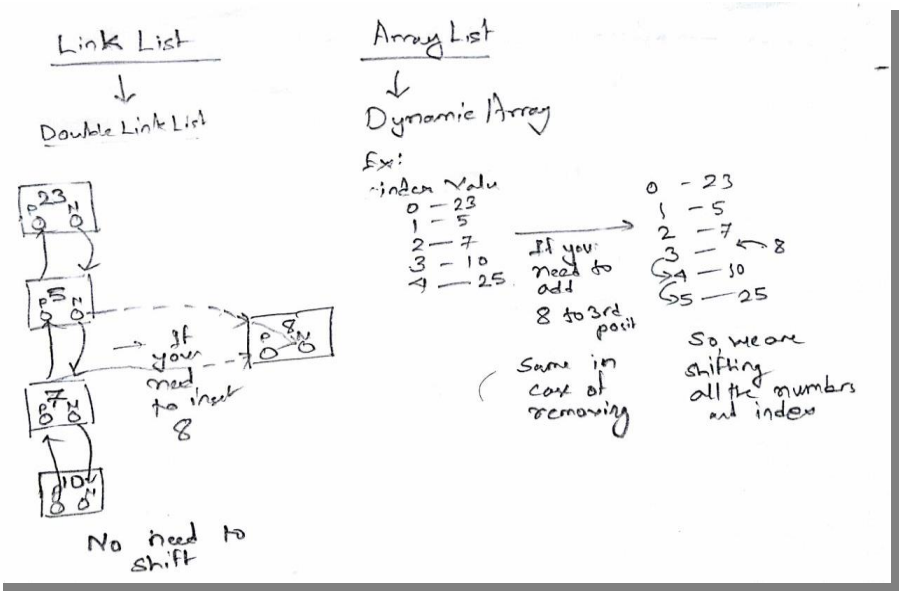
Differences between Vector And ArrayList:

	Vector	**ArrayList[Dynamic Array]**
When size is full Capacity Increases by	100%[Hence Lot of Memory Loss]	50%[Memory Efficient]
Synchronization	Thread Safe[At a time one Thread will work(Synchronized)]	Not Thread Safe[At a time multiple Thread will work(Un-Synchronized)]
Speed	Slow	fast

LinkedList and ArrayList:

- **Both are the implementation of List Interface**

LinkedList- Double Link List—Faster[When you want to assign the value in between]—Searching Slow[No Index No]
ArrayList-Dynamic Array—Slow[When you want to assign the value in between]—Searching Fast[As Index No is there]



```
List<Integer> list=new ArrayList<Integer>();
    list.add(305);
    list.add(998);
    list.add(774);
    list.add(236);
    list.add(881);

Collections.reverse(list); // Reverse the Collection by Index
Collections.sort(list);// This will sort the Array in Ascending order

list.forEach(System.out::println);
```

Now if you want to have a custom technique of Sorting, let suppose by the last digit of each number.
Then Comparator comes.

Note:

Two other Techniques:

```
List<Integer> listOther=Arrays.asList(56,24,33,25);

List<Integer> listOtherNew=new ArrayList<Integer>() {
      {
        add(23);
        add(45);
        add(76);
            }
      };
```

For Synchronizing an ArrayList:

```
Collections.synchronizedList(new ArrayList<>());
```

Comparator< Interface >:

Comparator Object has to be passed into the Collections.sort(). Comparator object will have the logic of Sorting.
A comparison function, which imposes a total ordering on some collection of objects. Comparators can be passed to a sort method (such as {@link Collections#sort(List,Comparator) Collections.sort} or {@linkArrays#sort(Object[],Comparator) Arrays.sort}) to allow precise control over the sort order.

```
     * @param o1 the first object to be compared.
     * @param o2 the second object to be compared.
     * @return a negative integer, if o1<o2
                 Zero, if o1=o2
                 a positive integer, if o1>o2
     * @throws NullPointerException if an argument is null and this
     *         comparator does not permit null arguments
     * @throws ClassCastException if the arguments' types prevent them from
     *         being compared by this comparator.


      Comparator comparator=new Comparator<Integer>() {
            @Override
            public int compare(Integer numFirst, Integer numSecond) {
                  if(numFirst%10>numSecond%10)
                        return 1;
                  return -1;//Also use Ternary Operator
                              return    numFirst%10>numSecond%10?1:-1;
            }
      };
      Collections.sort(list,comparator);// Comapared by last Digit
      list.forEach(System.out::println);
```

Output: `881  774  305 236 998`

Comparator with custom Type:

```
class Student{
	int rollnum;
	int marks;
	public Student(int rollnum, int marks) {
		super();
		this.rollnum = rollnum;
		this.marks = marks;
	}
	@Override
	public String toString() {
		return "Student [rollnum=" + rollnum + ", marks=" + marks + "]";
	}
}
```

List of Students:

```
List<Student> students=new ArrayList<Student>();
	students.add(new Student(1, 55));
	students.add(new Student(2, 35));
	students.add(new Student(3, 25));
	students.add(new Student(4,95));
	for (Student ss:students) {
		System.out.println(ss);
	}
```

```
Output:
Student [rollnum=1, marks=55]
Student [rollnum=2, marks=35]
Student [rollnum=3, marks=25]
Student [rollnum=4, marks=95]
```

The **Student** is a custom Class. We need to sort the list by the marks the Students.

```
Comparator<Student> comparator=new Comparator<Student>() {
@Override
public int compare(Student o1, Student o2) {
	return o1.marks>o2.marks?1:-1;
			}
		};
Collections.sort(students, comparator);
for (Student ss:students) {
	System.out.println(ss);
	}
```

```
Output:
Student [rollnum=3, marks=25]
Student [rollnum=2, marks=35]
Student [rollnum=1, marks=55]
Student [rollnum=4, marks=95]
```

Comparable<Interface>:

Now instead of passing a Comparator object to the Sort method, if the Student class knew itself how to sort its own objects.[`Collections.sort(students)`]

There comes the role of Comparable where you have to implement the interface Comparable in the level.

Where we have to implement the `compareTo(Student st)` method, which compares the current object and the next(Which we pass as parameter to the function) object.

```
class Student implements Comparable<Student>{
	int rollnum;
	int marks;
	public Student(int rollnum, int marks) {
		super();
		this.rollnum = rollnum;
		this.marks = marks;
	}
	@Override
	public String toString() {
		return "Student [rollnum=" + rollnum + ", marks=" + marks + "]";
	}
	@Override
	public int compareTo(Student st) {
		return this.marks>st.marks?1:-1;
	}
}
```

Then if you sort,

```
Collections.sort(students);
for (Student ss:students) {
  System.out.println(ss);
  }
```

```
Output:
Student [rollnum=3, marks=25]
Student [rollnum=2, marks=35]
Student [rollnum=1, marks=55]
Student [rollnum=4, marks=95]
```

Difference between Comparator and Comparable:

	Comparator<Interface>	Comparable<Interface>
	If we grow with **inbuilt class**[where we cannot change class definition] go for **Comparator**	If we grow with **custom class** go for **Comparable**
	Does not affect original class	Affects original class
Sorting Method	`Int compare(Object o1, Object o2)`	`compareTo(CustomClass st)`
Package	Java.util	Java.lang
Sorting	`Collections.sort(students, comparator);`	`Collections.sort(students);`
	Provide Multiple Sorting sequence	Provide single sorting sequence

Set<Interface>:

```
Set values=new HashSet();
```

HashSet

- It does not allow Duplicates.
- **HashSet** uses hashing concepts. Whenever we put this values into heap memory it goes into certain location. And hashing will use some algorithm using which the nearest value will come fast. HashSet will not give value in sequence.

TreeSet

Will provide sorted(Ascending Order) output.

Hashing Algorithm:

Map<Interface>

It's a key value pair.[Keys are always unique]

```
Map map=new HashMap();
map.put("myName", "King");
map.put("actor", "John");
map.put("ceo", "Rama");

System.out.println(map);
```

```
Output:
{actor=John, myName=King, ceo=Rama}
```

But you could see that the map elements are not displayed in order, similar to HashSet.
HashMap also uses hashcode and sorting Technique is hashing.

Displaying of Map:

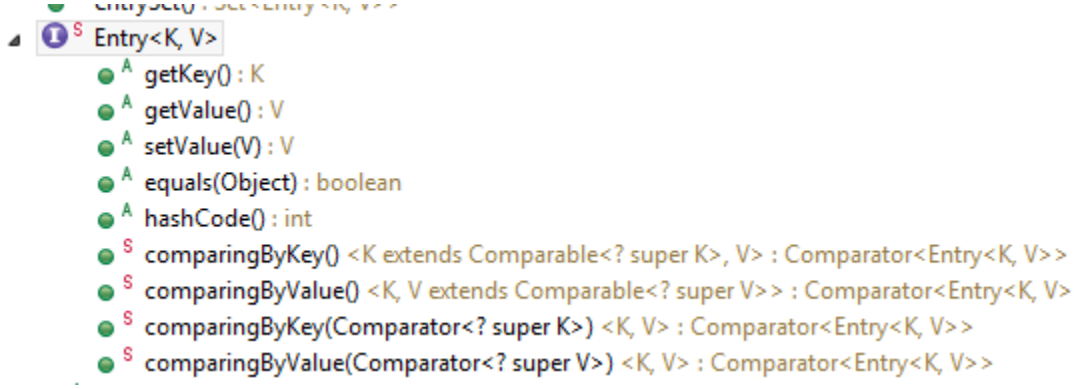
1. By Entry Set,

```
// We can get all the keys and store it in a set by the keySet function of Map
      Set<String> keys=map.keySet();
// Enhanced For Loop to iterate over each key
      for (String key:keys) {
      System.out.println(key+" : "+map.get(key));
      }
```

```
Output:
actor : John
myName : King
ceo : Rama
```

2. By Map Entry

Entry is an interface inside Map Interface.[Nested Interface]




```
//By HashTable[Hash Table is ThreadSafe(Synchronized)
Map phonebook=new Hashtable<>();
phonebook.put("King", "9002341234");// Each one is an entry
phonebook.put("Ram", "8017234567");
phonebook.put("Shyam", "9000135678");
phonebook.put("Rohit", "8888222209");

Set<Map.Entry<String, String>> values=phonebook.entrySet();
for (Map.Entry<String, String> entry:values) {
    System.out.println(entry.getKey()+":"+entry.getValue());
      }
```

```
Output:
Rohit:8888222209
Shyam:9000135678
Ram:8017234567
King:9002341234
```

HashTable:

It similar to HashMap, but it is synchronized[Thread Safe]

Differences between HashMap and HashTable

	HashMap	HashTable
	Introduced in 1.2 Version	It was there since Java was introduces
	Not Thread-safe[unsynchronized]	Thread-safe [Synchronized]
Speed	Fast	Slow
Threading	Works with single thread	Works with multiple threads
Null Key	Allows one null key	Does not allow null key

For Synchronizing Hash Map: `Collections.synchronizedMap(new HashMap<>());`

Similarity: Insertion Order is not maintained.

Linked HashMap: Insertion Order is Maintained.

Tree HashMap: Always Sorted

Advantages:
If Elements numbers are fixed always go for Arrays since array is fast compare to Collection.

Drawback:

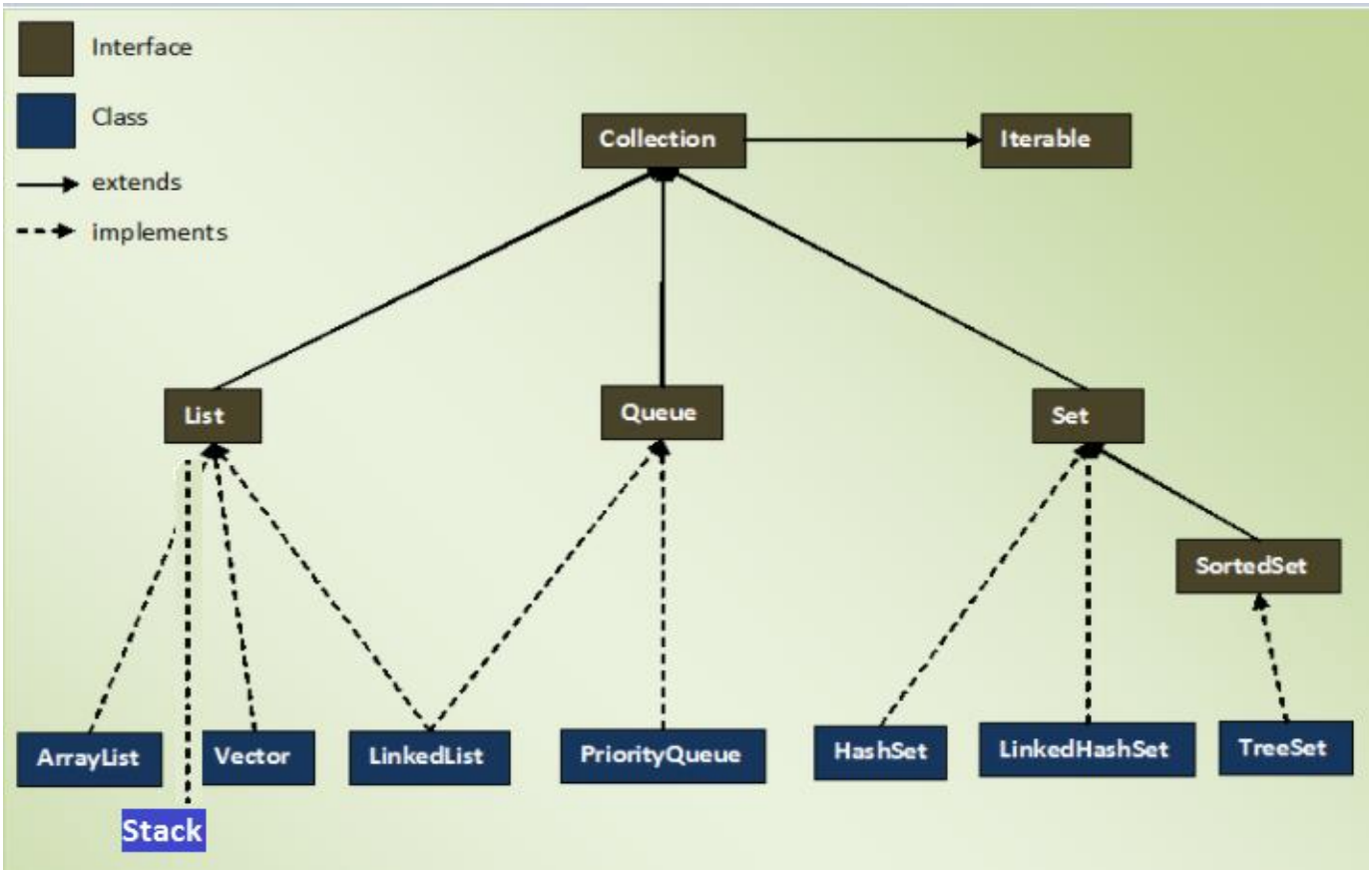
- Array can store only one type of data-type
- Inserting and deleting elements in the middle of the array is difficult.
- Size fixed

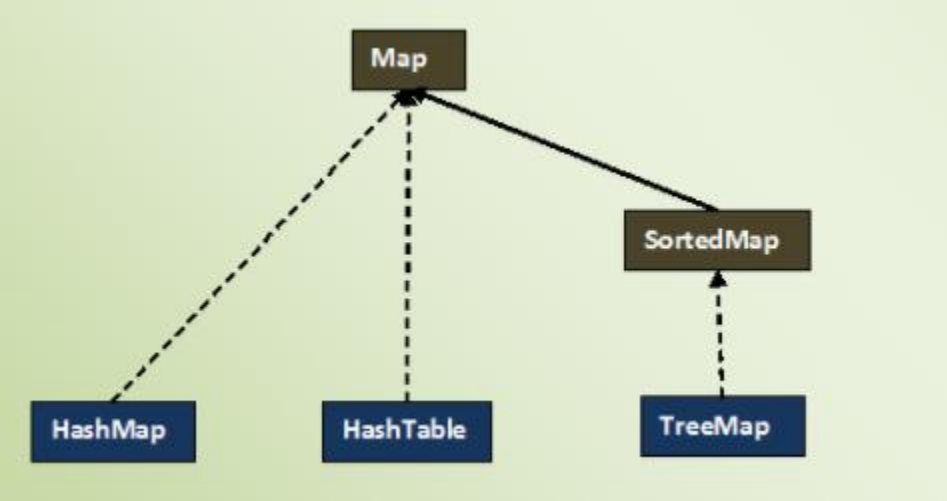
Q1. What is a collection framework?
Ans: A collection framework is a class library to handle group of objects. Collection Framework is implemented in **java.util** package
A collection object or an container object is an object which can store a group of other objects.

Q2. What is a **Collections**?
Ans: **Collections** is an Class which is present inside the **java.util** package.
It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection.

- Java Collection class supports the **polymorphic algorithms** that operate on collections.
- Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

Q3. Does collection object store copies of other objects or their references?
Ans: Collection object does not store the physical copies of other object, it simply store the references of other objects into a collection object.





Q4. Can you store a primitive datatype into a collection?
Ans: No, collection can only store objects.

Q5. How can we retrieve elements from collections?
Ans: Following are the 4 ways to retrieve any elements from a collection object:

- Using **for-each** loop
- Using **Iterator** Interface
- Using **ListIterator** interface
- Using **Enumeration** Interface

For-each Loop:
It's like a for loop only which repeatedly executes a group of statements for each element of the collection.

```
for (Object object : value) {
   //Statements
         }
```

Iterator Interface:

This interface contains methods to retrieve elements from the collection object only in forward direction.

- **boolean hasNext() :**
- **element next() :**
- **void remove()** :The method removes from the collection the last element returned by the iterator.

ListIterator interface:

This interface contains methods to retrieve elements from the collection object both in forward and reverse direction.

The methods of the interface are as follows:

- **boolean hasNext()**
- **boolean hasPrevious()**
- **element next()**
- **element previous()**
- **void remove()**

Q6. What is the difference between iterator and ListIterator?
Ans: Both are useful to retrieve elements from collection. Iterator can retrieve elements only in the forward direction. But ListIterator can retrieve the elements in the forward and backward direction also. So, ListIterator is preferred over Iterator.

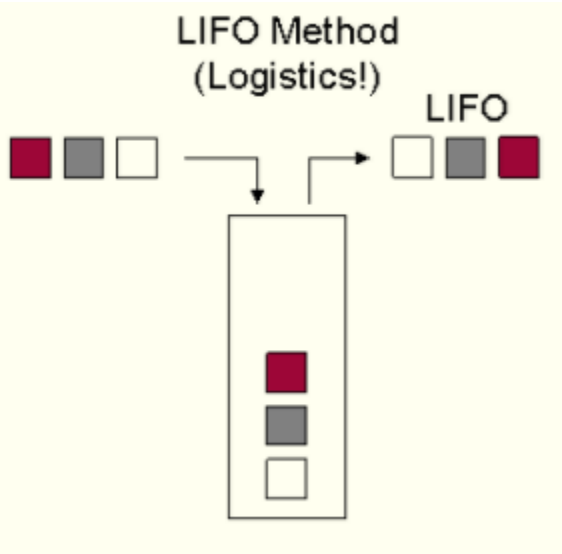
Enumeration Interface:
Methods of Enumeration Interface,

- boolean hasMoreElements()
- element nextElement()

HashSet :

- **Does not guarantee the order of element**
- **No Duplicate Elements.**

Q7. What is Stack? What are its methods?
Ans: A stack represents a group of elements stored in LIFO(Last In First Out)




```
Stack<Integer> stack=new Stack<Integer>();
stack.isEmpty();//Out=boolean/If Stack empty then True else False
stack.peek();//Out=Element/Returns Top Most Object without removing it
stack.pop();//Out=Element/pops the top most element and returns it
stack.push(23);//Out=Element/pushes an element on the top of the stack and returns that
                        element
stack.search(45);//Out=int/Returns the position of the element from the top of the stack
                                                //If object is not found then returns -1.
```

Q8. What is autoboxing?
Ans: Converting a primitive datatype into an object form automatically is called 'auto boxing'. Autoboxing is done in generic types.

Q9. What is the differences between Stack and LinkedList?
Ans:

- A stack is generally used for evaluation of expression whereas LinkedList is used to store and retrieve data.
- Insertion and Deletion of element only from the top of the Stack is possible whereas in case of LinkedList it is possible anywhere.

Q10. What is default capacity of a HashMap? Ans: 16
Q11. What is load Factor?

Ans: Load Factor Represents at what level the HashMap Capacity should be doubled.
Example: LoadFactor of HashMap or HashTable is 0.75