Collections and Generics

$$\begin{tikzcd}
 & \text{Java.util package} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Legacy classes(}\textbf{jdk1.0}\text{)} & \text{Collection Framework(}\textbf{jdk1.2}\text{)} & \text{Generic types(}\textbf{jdk1.5}\text{)}
\end{tikzcd}$$

Limitations with arrays:

1) Size of an array is fixed i.e., once an array is created, there is no way of increasing or decreasing its size.
2) Array can hold only homogeneous data elements.
   Example:
   ```
   String[] str = new String[5];
   Str[0] = new String("aspire");
   Str[1]=new Integer(10); //C.E
   ```
   But we can partially overcome this limitation by using Generic array (Object[]).
   ```
   Object[] arr = new Object[5];
   arr[0] = new String("aspire");
   arr[1]=new Integer(10);
   ```
3) No predefined data structors to manage array data.

To overcome above limitations, java.util package was added to J2SE:

1) The size of a collection is not fixed i.e., the size of a collection can be increased or decreased.
2) Collection can hold heterogeneous (any type of) objects.
3) Predefined data structors such as Stack, Queue, DeQue, LinkedList,etc are defined to manage collection data.

Difference between array and collection:

Array	**Collection**
Size is fixed	Collection size Can grow or shrink
Contains only homogeneous elements.	Contains heterogeneous elements.
The elements type can be Primitive or Reference type.	Allows only Reference types but not Primitive type.
No predefined data structors to manage array data.	Predefined data structors are defined to manage collection data.

Collection Framework

It defines group of classes and interfaces. Collection can be used to represent group of objects as a single entity (single object).

Collection Framework defines the following 9 key interfaces

1. Collection <<interface>>

It represents group of individual objects as a single entity (object).
It acts as a root interface for entire collection framework.

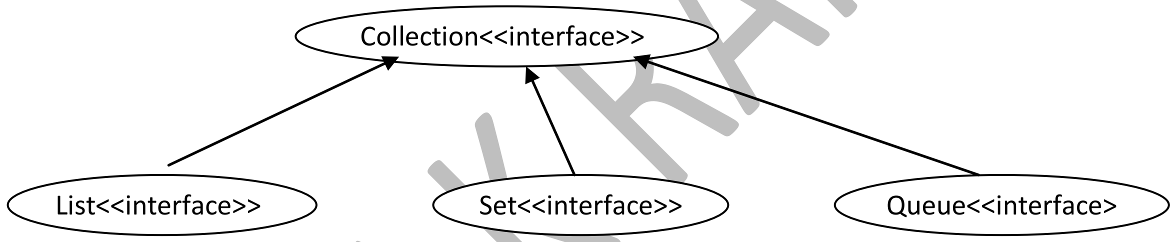
This interface defines the most common methods, which can be applied to any collection object.
There is no direct subclass which implements Collection interface directly.

Collection Vs Collections vs collection
Collection is an interface. Where as Collections is a class, which contains utility methods such as sort(), binarySearch(), copy(), etc, and collection represents group of objects as a single entity.

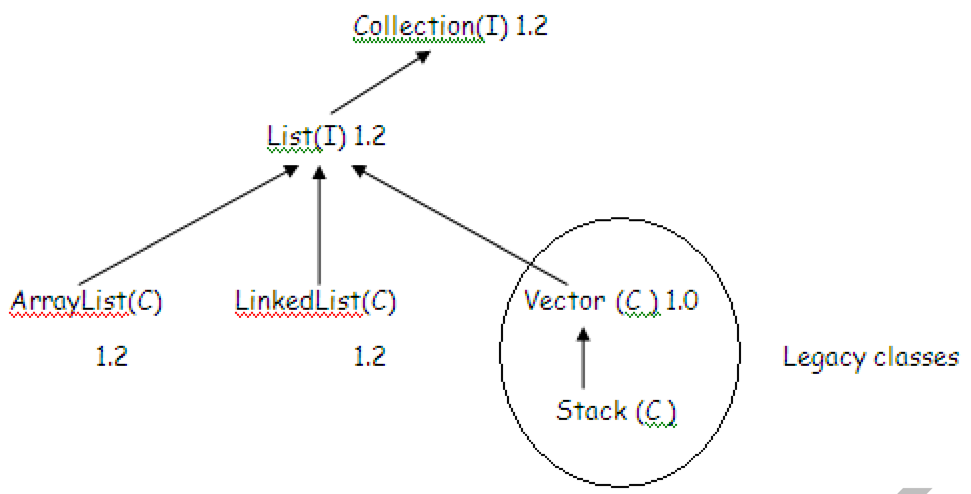
Collection properties:

1. Some Collection implementation classes allow **duplicate** elements and others will not allow duplicate elements.
2. Some Collection implementation classes are **ordered** and others are unordered collections.
3. Some Collection implementation classes allow **null values** and others will not allow null values.
4. By default, elements in some of the collection implementation classes are **sorted** and others are not sorted.
5. By default, elements in some of the collection implementation classes are **synchronized** and others are not synchronized.

There are 3 direct subinterfaces of Collection interface:

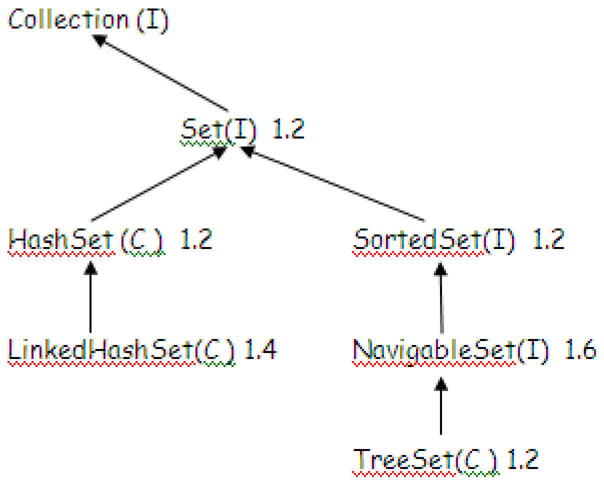


2. List <<interface>>
It is inherited from a Collection interface.
It allows duplicate elements.
List is an ordered collection based on index but not insertion.
List allows more than one null value.
By default, none of the List implementation classes are sorted.
By default, some of the List implementation classes are synchronized.



3. Set <<interface>>

It is inherited from Collection interface.
Set does not allow duplicate elements.
The Set elements order depends on its implementation class.
Some Set impl classes will not allow null values but others may allow at most one null value.
By default, some of the Set implementation classes are sorted and others are not.
By default, none of the Set implementation classes are synchronized.



4. SortedSet <<interface>>

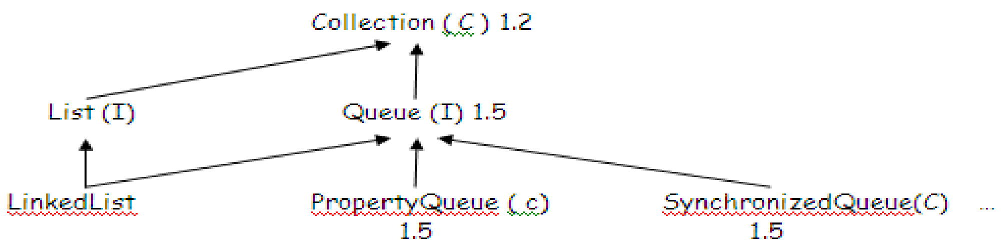
It is inherited from Set interface.
Elements are sorted using natural or customized sorting order.

5. NavigableSet <<interface>>

It is inherited from SortedSet interface, hence elements are sorted in natural or customized sorting order.
It defines several new methods to support navigation in TreeSet.

6. Queue <<interface>>

Queue is inherited from Collection interface.
The order of Queue elements depends on its implementation classes.
For LinkedList, the elements order is FIFO.
For PriorityQueue,the elements order is natural or customized sorting order.



Note: In the above diagram, class name is **SynchronousQueue** but not SynchronizedQueue.
In the above diagram, class name is **PriorityQueue** but not PropertyQueue.
From jdk1.5 version onwords LinkedList implements Queue Interface as well.

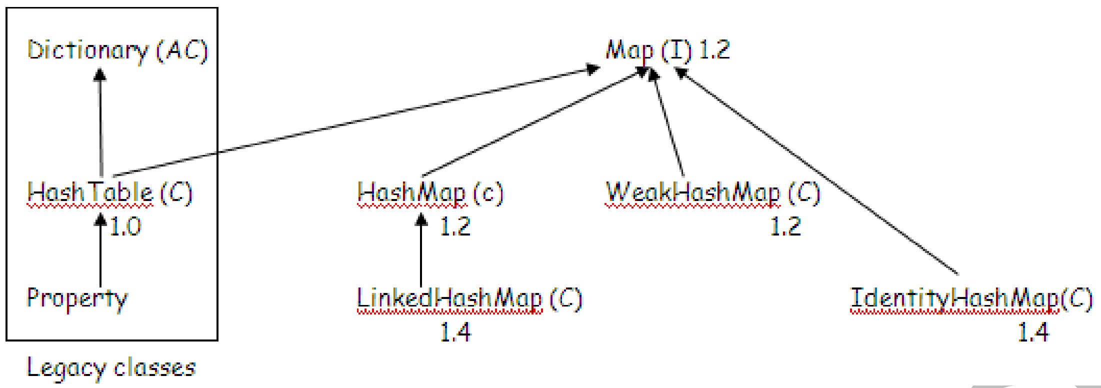
All above interfaces are meant for representing individual objects only.
If we want to represents a group of objects as key & value pairs, then we should go for Map interface.
Example:

```
driverClass=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:XE
username=scott
password=tiger
```

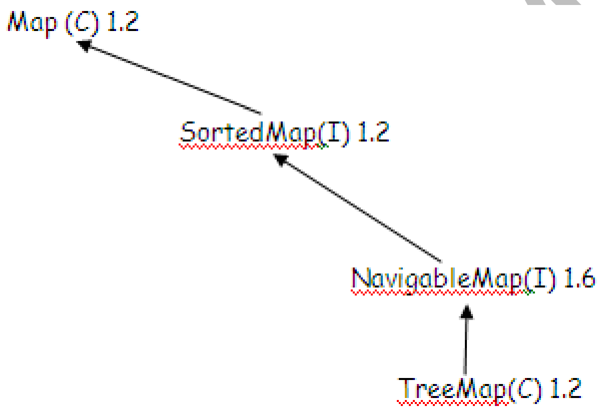
Key	**Value**
driverClass	oracle.jdbc.driver.OracleDriver
url	jdbc:oracle:thin:@localhost:1521:XE
Username	Scott
Password	Tiger

7. Map <<interface>>

It represent group of key and values pairs i.e., group of entries.
This is the root interface for all key and value based collections.
Map interface is not inhereited from Collection interface.
Keys are never duplicated, but values can be duplicated.
Some of the Map impl classes are ordered based on key.
Some of the Map impl classes allows both null key and null values.
By default, some of the Map impl classes are sorted based on key.
By default, some of the Map impl classes are synchronized.

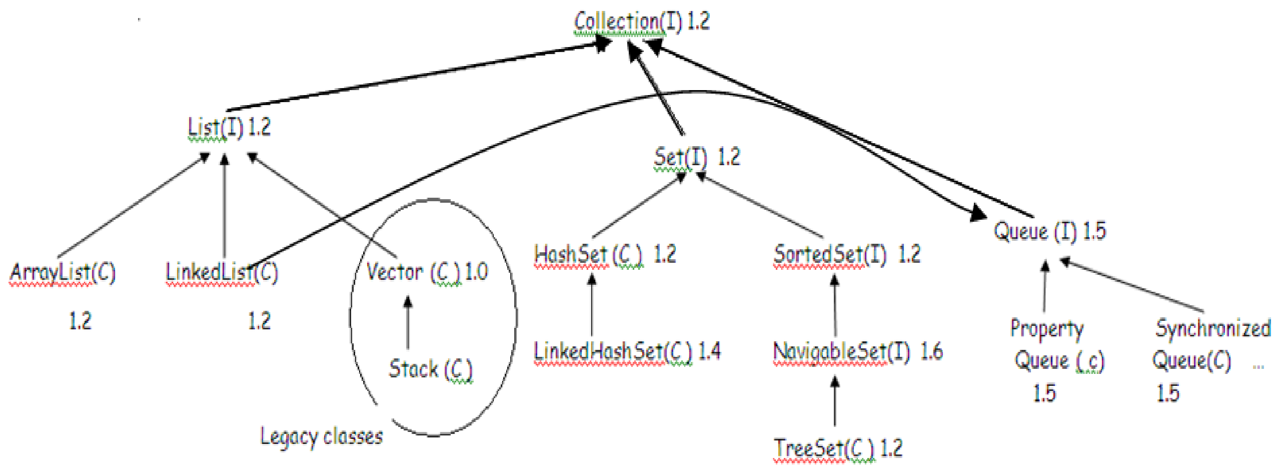


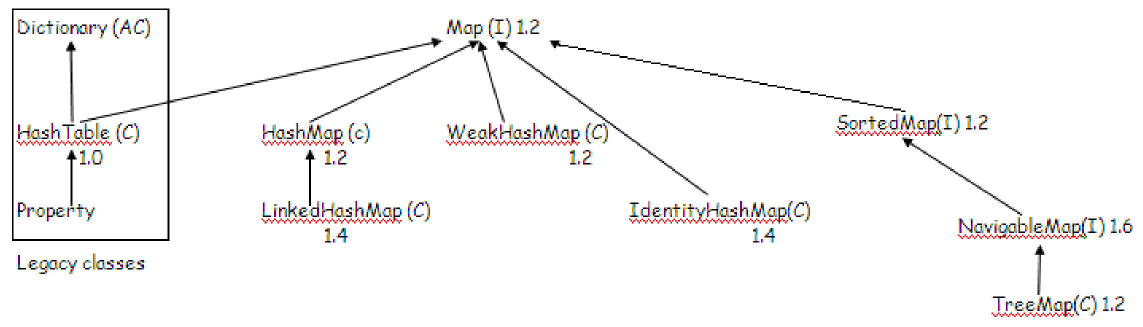
8. **SortedMap <<interface>>**
   It is inhered from Map interface.
   Entries are inserted according to natural or customized sorting order based on keys.



9. **NavigableMap <<interface>>**
   This is inherited from SortedMap interface.
   Defines several new methods to support the navigation in TreeMap.

Summery





The legacy classes/interfaces in java.util package are:

1. Vector
2. Stack
3. Hashtable
4. Properties
5. Dictionary <<abstract class>>
6. Enumeration <<interface>>

By default, all legacy classes are synchronized means thread-safe.

Collection <<interface>>:

It is a root interface having most common methods which are applicable for all collection implementation classes.

Basic operations in collections are:

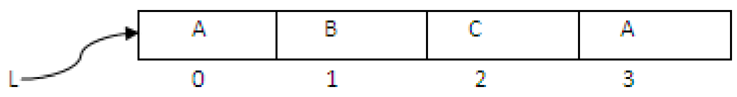
1. Add element(s) to the collections.
2. Remove element(s) from the collection.
3. Search element(s) in the collection.
4. Retrieve all elements from the underlying collection (Iterate through the collection).
5. Peform all above operations but based on index, hence it is applicable only for List implementation classes.

To perform above operations, the Collection interface have the following common methods:

1. Boolean add(Object o);
2. Boolean addAll(Collection c); } //Add object(s) to the collection.
3. Boolean remove(Object o);
4. Boolean removeAll(Collection c); } //Remove object(s) from the collection.
5. Void clear();
6. Boolean contains(Object o);
7. Boolean containAll(Collection c); } //Search object(s) from the collection.
8. Iterator iterator(); //Iterate through the collection.
9. Boolean isEmpty();
10. Int size();
11. Object[] toArray();

List <<inteface>>

Since List is an ordered collection based on index, List interface have index specific methods along with common methods inherited from Collection interface.



1. Boolean add(int index, Object o);
2. Boolean addAll(int index, Collection c);
3. Object remove(int index);
4. Int indexOf(Object o); //Returns index of first occurance of specified object, otherwise returns -1
5. Int lastIndexOf(Object o); //Returns index of last occurance of specified object, otherwise returns -1
6. Object set(int index, Object new); //Replaces with new element
7. ListIterator listIterator(); //Applicable only for List implementation classes.
8. Object **get**(int index); //Retrieves an element from the specified index.

Vector

Vector is a growable array whose size can be grown or shrink.
Vector allows duplicate elements.
Vector is an ordered collection based on index.
Vector allows null values.
By default, vector elements are not sorted.
Vector is a synchronized collection (thread-safe), since it is a legacy class.

Example:

```
//VectorDemo.java
import java.util.Enumeration;
import java.util.Vector;
public class VectorDemo{
        public static void main(String[] args) {
                Vector v = new Vector();
                v.addElement(1); // auto boxing
                v.addElement(new Integer(5));
                v.add(new Integer(3));
                v.add(new Integer(3));   //allows duplicate element
                v.add(null);        //allows null value
                v.add(null);        //duplicate null value
                v.add(0, new Integer(4)); //adding element using index position
                v.add("hello");
                System.out.println(v.toString()); //[4, 1, 5, 3, 3, null, null, hello]
        }
}
```

Iterating through collection elements

Used to read (or retrieve) all elements but one at a time from the underlying collection.
There are 4 ways in which objects can be retrieved from the underlying collection:

1. Enumeration : Applicable only for legacy classes.
2. Iterator : Applicable for all Collection implementation classes.

3. ListIterator : Applicable only for List implementation classes.
4. Enhanced for loop : In case of Generic types

Enumeration <<Interface>>

Introduced in 1.0 version. Hence, it is applicable only for legacy classes.
The following method is applicable only for legacy classes, whose return type is an Enumeration.

public Enumeration elements()

Enumeration interface defines the following two methods

1. Public Boolean hasMoreElements();
2. Public Object nextElement();

Example: Print even numbers

```
import java.util.*;
public class EnumerationDemo{
        public static void main(String[] args){
                Vector v= new Vector();
                for(int i=1; i<=10; i++){
                        v.addElement(i);
                }
                System.out.println(v);
                Enumeration e=v.elements();
                while(e.hasMoreElements()){
                        //int  i= e.nextElement(); //C.E. Downcasting is not implicit.
                        int  i=(Integer)e.nextElement();
                        if(i%2==0){  //print even numbers
                                System.out.print(i+"\t");
                        }
                }
        }
}
```

Limitations with Enumeration:

1) Enumeration is applicable only for legacy classes.
2) While iteratering, we can perform only read operation, but we cannot remove or modify an element from the underlying collection.

Iterator

It is introduced in 1.2 version. Hence, it is used for legacy and non-legacy classes.
The following method is declared in the Collection interface, hence it is applicable for all Collection implementation classes.

Public Iterator **iterator**()

Iterator interface defines the following three methods:

1. Public Boolean hasNext();
2. Public Object next();
3. Public void remove();

Example: Print even numbers and remove odd numbers

```
import java.util.*;
public class IteratorDemo{
        public static void main(String[] args){
                Vector v=new Vector();
                for(int i=1;i<=10; i++){
                        v.add(i);
                }
                System.out.println(v);//[1, 2, 3, 4, 5, 6, 7, 8,9, 10]
                Iterator itr=v.iterator();
                while(itr.hasNext()){
                        Integer i=(Integer)itr.next();
                        if(i%2 != 0){ //odd numbers
                                itr.remove();
                        }
                }
                System.out.println(v);//[2, 4, 6, 8, 10]
        }
}
```

Advantages with Iterator

1) Applicable for both non-legacy classes along with legacy classes.
2) Can remove element from the underlying collection.
3) Method names are improved.

Limitations with Enumeration and Iterator:

1. The main limitation with enumeration and iterator is, they can move to forward direction only.
2. We can perform read and remove operations but not add and modify operations.

ListIterator <<inteface>>

It is the child interface of Iterator, it has introduced in 1.2 version.
ListIterator is applicable only for List implementation classes.
The List interface have the following method, whose return type is ListIterator:

Public ListIterator **listIterator**()

This interface defines the following 9 methods.

1. Public Boolean hasNext();
2. Public Boolean hasPrevious();
3. Public Object next();
4. Public Object previous();
5. Public int nextIndex();
6. Public int previousIndex();
7. Public void remove();
8. Public void set(Object O);
9. Public void add(Object new); // We can add new object to the underlying collection.

Example:

```
import java.util.*;
public class ListIteratorDemo{
	public static void main(String[] args){
		List players=new Vector();
		players.add("Dhoni");
		players.add("Kohli");
		players.add("Sachin");
		players.add("Raina");
		System.out.println(players);//[Dhoni, Kohli, Sachin, Raina]

		ListIterator itr= players.listIterator();
		while(itr.hasNext()){
			String player=(String)itr.next();
			if(player.equals("Sachin")){
				/*itr.remove();
				itr.add("Rohit");*/
				itr.set("Rohit");
			}
		}
		System.out.println(players);//[Dhoni, Kohli, Rohit, Raina]
	}
}
```

Note:
Among all three iterator classes, ListIterator is most powerfull iterator class. But the only limitation with this interface is, it is applicable only for List implementation classes.

Comparison table for Enumeration, Iterator, and ListIterator

Property	Enumeration	Iterator	ListIterator
Is legacy?	Yes	No	No
It is applicable for	Only legacy classes	Both legacy and no-legacy classes	Only for List implementation classes
Navigation	Forward direction	Forward direction	Bi-directional
How to get	Using elements() method	Using iterator() method	Using listIterator() method
Operations	Only read	Read and remove	Read, remove, add, modify
Methods	hasMoreElements(), nextElements()	hasNext(); next(), remove()	9 methods.

Sorting List Elements

The Collections class provides following methods to sort List elements explicitly:

Public static void sort(List) // For Natural Sorting Order

Public static void sort(List, Comparator) //For Customized Sorting Order

To sort list elements, use one of the following interfaces:

a) Java.lang.Comparable
b) Java.util.Comparator

Comparable <<interface>>

It is used to provide **natural sorting** order.

This interface is available in java.lang package.

This interface contains only one method which is **compareTo()**.

public int compareTo(Object O)

o1.compareTo (o2);

Return –ve means o1<o2

Return +ve means o1>o2

Returns 0 means o1 == o2 means equal

All wrapper and String classes are inherited from Comparable interface.

Example #1: Natural sorting order for Built-In String class.

```
import java.util.Collections;
import java.util.Comparator;
import java.util.Vector;
public class SortDemo1{
        public static void main(String[] args) {
                List fruits = new Vector();
                fruits.add("Grapes");
                fruits.add("Banana");
                fruits.add("Papaya");
                fruits.add("Apple");

                System.out.println(fruits.toString());[Grapes, Banana, Papaya, Apple]

                Collections.sort(fruits); //Natural sorting order
                System.out.println(fruits.toString()); [Apple, Banana, Grapes, Papaya]
        }
}
```

Example #2: Natural sorting order for user defined classes.

In case of natural sorting order, JVM automatically invokes compareTo() method.

```
import java.util.Collections;
import java.util.List;
import java.util.Vector;
class Student implements Comparable{
	private int sno;
	private String sname;
	public Student() {}
	public Student(int sno, String sname){
		this.sno = sno;
		this.sname = sname;
	}

	@Override
	public int compareTo(Object obj){
		Student sRef = (Student)obj;
		Integer iRef1 = this.sno;
		Integer iRef2 = sRef.sno;

		return iRef1.compareTo(iRef2);
	}

	@Override
	public String toString(){
		return this.sno+" "+this.sname;
	}
}
public class SortDemo2 {
	public static void main(String[] args) {
		List students = new Vector();

		Student s1 = new Student(2, "xyz");
		Student s2 = new Student(1, "abc");
		Student s3 = new Student(3, "pqr");

		students.add(s1);
		students.add(s2);
		students.add(s3);

		Collections.sort(students); //Natural sorting order
		System.out.println(students);
	}
}
```

Limitations with Comparable interface:

1) We cannot provide multiple sorting techniques for both built-in and user defined classes.
2) We cannot provide sorting order for non-comparable built-in classes such as StringBuffer.
3) We cannot provide different sorting order other than natural sorting order for built-in comparable classes such as String, Integer, etc.
4) We cannot sort heterogeneous elements using Comparable interface.

Java.util.Comparator <<interface>>

This interface is available in java.util package.

Comparator interface is used to provide **customized sorting** order.

The customized sorting order is required for following scenarios:

1) To provide multiple sorting techniques for both built-in and user-defined classes.
   Example:
   a) Sort students based on roll number [AND]
   b) Sort students based on student name [AND]
   c) Sort students based on student age.
2) To provide sorting order for non-comparable built-in classes such as StringBuffer i.e., built-in classes which are not inherited from Comparable interface.
   For Example: The StringBuffer class is not inherited from Comparable interface, hence java.util.Comparator interface must be used to sorting stringbuffer elements.
3) To provide different sorting techniques other than natural sorting order for built-in Comparable classes.
   Example:
   a) Sort strings in descending order.
   b) Sort integers in descending order.
4) To sort Heterogeneous elements.
   For example, if the collection contains both String and StringBuffer objects.

Methods:

The Comparator interface contains the following two methods:

1. Public int **compare**(Object o1, Object o2)
   Return –ve means o1<o2
   Return +ve means o1>o2
   Returns 0 means o1 == o2 means equal
2. Public Boolean equals(Object obj);

The subclass of Comparator interface must implement compare() method.

The subclass of Comparator interface optionally implement equals() since it is already available in Object class.

Case 1: Provide multiple sorting techniques for both built-in and user defined classes

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Vector;
class Employee{
        private int eno;
        private String ename;
        public Employee() {       }
        public Employee(int eno, String ename){
                this.eno = eno;
                this.ename = ename;
        }
        public int getEno(){return this.eno;}
        public String getEname(){return this.ename;}

        @Override
        public String toString(){
                return this.eno+" "+this.ename +"\t";
        }
}

//sort based on employee number
class EnoSort implements Comparator{
        @Override
        public int compare(Object o1, Object o2){
                Employee e1 = (Employee)o1;
                Employee e2 = (Employee)o2;

                Integer iRef1 = e1.getEno();
                Integer iRef2 = e2.getEno();

                return iRef1.compareTo(iRef2);
        }
}

//sorting based on employee name
class EnameSort implements Comparator{
        @Override
        public int compare(Object o1, Object o2){
                Employee e1 = (Employee)o1;
                Employee e2 = (Employee)o2;

                String str1 = e1.getEname();
                String str2 = e2.getEname();

                return str1.compareTo(str2);
        }
}

public class SortDemo3 {
        public static void main(String[] args) {
                List emps = new Vector();
                Employee e1 = new Employee(2, "xyz");
                Employee e2 = new Employee(1, "abc");
                Employee e3 = new Employee(3, "pqr");

                emps.add(e1);
                emps.add(e2);
                emps.add(e3);

                Collections.sort(emps, new EnoSort()); //Customized sorting order based on eno
                System.out.println(emps.toString());

                Collections.sort(emps, new EnameSort()); //Customized sorting order based on ename
                System.out.println(emps.toString());
        }
}
```

Case 2: Provide sorting order for non-comparable built-in classes such as StringBuffer

Example:

```
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.Vector;
class MyComparator implements Comparator{
        @Override
        public int compare(Object o1, Object o2) {
                String str1 = o1.toString();
                String str2 = o2.toString();
                return str1.compareTo(str2);
        }
}
public class ComparatorDemo1 {
        public static void main(String[] args) {
                StringBuffer sb1 = new StringBuffer("A");
                StringBuffer sb2 = new StringBuffer("C");
                StringBuffer sb3 = new StringBuffer("B");

                List v = new Vector();
                v.add(sb1);
                v.add(sb2);
                v.add(sb3);

                // Collections.sort(v); //throws C.C.E since collection elements are non comparable.
                Collections.sort(v, new MyComparator());
                System.out.println("After sorting....");
                System.out.println(v.toString()); //[A, B, C]
        }
}
```

O/P:

```
After sorting....
[A, B, C]
```

ClassCastException w.r.t Collections.sort() method:

I. The Collections.sort(List) utility method throws ClassCastException when collection elements are non comparable.

Example:

```
StringBuffer sb1 = new StringBuffer(“one”);
StringBuffer sb2 = new StringBuffer(“two”);
List v = new Vector();
v.add(sb1);  v.add(sb2);
Collections.sort(v);        //throws C.C.E
```

II. The Collections.sort(List) utility method throws ClassCastException even though elements are comparable but not compatible.

Example:

```
String str = new String(ten);
Integer iRef = 10;
List v = new Vector();
v.add(str);
v.add(iRef);
Collections.sort(v);  // throws C.C.E
```

Overall scenarios which likely to throw ClassCastException:

1) Assigning supertype object to subtype reference variable.
2) Unrelated object and reference types.
3) Non comparable classes such as StringBuffer for sorted collections.
4) Non homogeneous comparable elements in sorted collections.

Case 3: Provide sorting technique other than natural sorting order for built-in comparable classes.

Example#1: Sort string elements in descending order.

```
//Descending order
class StringDescendingOrder implements Comparator{
	public int compare(Object o1, Object o2) {
		String  str1 = o1.toString();
		String   str2 = o2.toString();
		return –str1.compareTo(str2);
	}
}
//Client code
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Vector;
public class ComparatorDemo3 {
	public static void main(String[] args) {
		String s1 = "A";
```

```
        String s2 = "C";
        String s3 = "B";

        List list = new Vector();
        list.add(s1);
        list.add(s2);
        list.add(s3);

        Collections.sort(list); //Natural sorting order
        System.out.println("Ascending Order:");
        System.out.println(list.toString());         //O/P:   [A, B, C]

        Collections.sort(list, new StringDescendingOrder()); //Descending order
        System.out.println("Descending Order:");
        System.out.println(list.toString());         //O/P:  [C, B, A]
    }
}
```

Example#2: Sort integer elements in descending order

```
//Descending order
class IntegerDescendingOrder implements Comparator{
    public int compare(Object o1, Object o2) {
        Integer iRef1 = (Integer)o1;
        Integer iRef2 = (Integer)o2;

        //return –iRef1.compareTo(iRef1);
        return iRef2.compareTo(iRef1);
    }
}
```

//Client code

```
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Vector;
public class ComparatorDemo4 {
    public static void main(String[] args) {
        Integer iRef1 = 20;
        Integer iRef2 = 10;
        Integer iRef3 = 30;

        List list = new Vector();
        list.add(iRef1);
        list.add(iRef2);
```

```
                list.add(iRef3);

                Collections.sort(list);//Natural sorting order
                System.out.println("Natural sorting order...");
                System.out.println(list.toString());        //[10, 20, 30]

                Collections.sort(list, new IntegerDescendingOrder());//Descending order
                System.out.println("Descending order...");
                System.out.println(list.toString());        //[30, 20, 10]
        }
}
```

If we implement compare() method in the following way then the corresponding outputs are:

```
            public int compare(Object Obj1, Object Obj2){
                Integer I1=(Integer) Obj1;
                Integer I2=(Integer) Obj2;
```

Note1: Return I1.compareTo(I2); ascending order [10,20,30]
Note2: Return -I1.compareTo(I2); descending order [30,20,10]
Note3: Return I2.compareTo(I1); descending order [30,20,10]
Note4: Return -1; just insertion order [20, 10, 30]
Note5: Return 1; reverse of insertion order [30, 10, 20]
Note6: Return 0; the TreeSet ultimately contains 1 obj irrespective of number of objects added

Case 4: Sort heterogeneous elements using Comparator interface.

Write a program to insert string and stringbuffer objects into the Vector, where the sort order is increasing string length order. If two string are having the same length then use the alphabetical order.

Example:

```
import java.util.*;
class LengthComparator 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 if (l1>l2)
                        return +1;
                else
                        return s1.compareTo(s2);
        }
}
class ComparatorDemo5{
        public static void main(String[] args) {
                List t=new Vector();
                t.add(new StringBuffer("ABC"));
                t.add(new StringBuffer("AA"));
                t.add("XX");
                t.add("ABCD");
                t.add("A");
                Collections.sort(t, new LengthComparator());
                System.out.println(t);//[A, AA, XX, ABC, ABCD]
        }
}
```

Comparison between Comparable and Comparator

Comparable	Comparator
Present in java.lang package	Present in java.util package
Contains only one method compareTo(Object);	Contains two methods Compare(Object, Object), and equals().
This interface is for defining natural sorting order.	This is for defining Customized sorting order.
By default, some of the built-in classes are inherited from Comparable interface.	None of the built-in classes are inherited from Comparator interface.
It is a marker interface	It is not a marker interface
We can provide only one sorting technique.	**We can provide multiple sorting techniques.**

ArrayList

ArrayList is a growable array i.e., its size can be increased or decreased.
ArrayList allows duplicate elements.
ArrayList is an ordered collection based on index.
ArrayList allows duplicate null values.
By default, ArrayList elements are not sorted. But, if required, ArrayList elements can be sorted explicitly using:

Collections.sort(List) : For natural sorting order
Collections.sort(List,Comparator) : For Customized Sorting order

By default, ArrayList is non-synchronized collection (non thread-safe). To obtain **Synchronized version of ArrayList**, use following utility method from Collections class:

Public static List synchronizedList(List l);

Example

```
ArrayList aList=new ArryList();// aList is non-synchronized.
List l=Collection.synchronizedList(aList); // aList becomes Synchronized now
```

Similarly we can get Synchronized version of Set and Map by using following utility methods from **Collections** class:

Public static Set synchronizedSet(Set s);
Public static Map synchronizedMap(Map m);

Constructors

1. **ArrayList al=new ArrayList()**

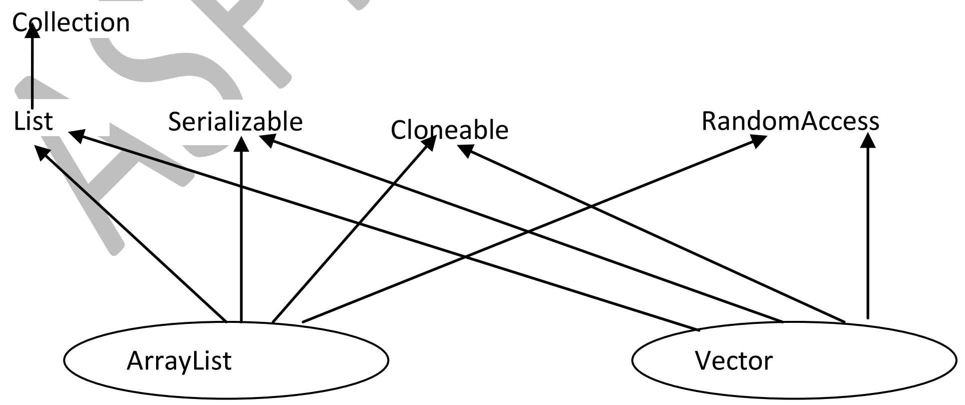
Creates an Empty ArrayList object with default initial capacity is 10. When the maximum capacity reaches then new ArrayList object is created.

2. **ArrayList al= new ArrayList(int initial_capacity)**
   Creates an empty ArrayList object with specified initial capacity.
3. **ArrayList al=new ArrayList(Collection c)**
   Creates an ArrayList object and initializes with specified collection elements.

Example:

```
import java.util.*;
class  ArrayListDemo{
        public static void main(String[] args){
                ArrayList al=new ArrayList();
                al.add("A");
                al.add(10);
                al.add("A");
                al.add(null);
                System.out.println(al);//[A, 10, A, null]
                al.remove(2);
                System.out.println(al); //[A, 10, null]
                al.add(2,"M");
                al.add("N");
                System.out.println(al);// [A, 10, M, null, N]
        }
}
```

Usually collection data is transfered from one JVM instance to other JVM instance. To support this requirement, every collection class must be inherited from **java.io.Serializable** interface.
Also, collection data can be copied. Hence, every collection class must be inherited from **java.lang.Cloneable** interface.
ArrayList and Vector classes implements **java.util.RandomAccess,** which is marker interface, to indicate that they support constant access time.



Example:
ArrayList al=new ArrayList()
LinkedList ll=new LinkedList();

```
System.out.println(al instanceof Serializable);//true
System.out.println(ll instanceof Cloneable);//true
System.out.println(al instanceof RandomAccess);//true
System.out.println(ll instanceof RandomAccess);//false
```

Difference between ArrayList and Vector:

ArrayList	Vector
Non synchronized collection i.e., non thread-safe.	By default, synchronized i.e., thread-safe.
High Performance.	Low Performance.
Non-legacy class and introduced in 1.2 version	Legacy class and introduced in 1.0 version

Recommended
ArrayList and Vector are best choice for frequent retrieval, because ArrayList and Vector supports random access.

Not recommended
To insert or delete an element from the middle, lot of elements needs to be shuffled (moved). Hence, ArrayList and Vector are not recommended for frequent insertions or deletions in the middle.

LinkedList

1. The underlying data structure is doubly LinkedList.
2. LinkedList allows duplicate elements.
3. LinkedList is an ordered collection based on index but not insertion.
4. LinkedList allows duplicate null values.
5. By default, LinkedList elements are not sorted. But, if required, LinkedList elements can be sorted explicitly using:
   Collections.sort(List) : For natural sorting order
   Collections.sort(List,Comparator) : For Customized Sorting order
6. By default, LinkedList is non-synchronized class (non thread-safe). To obtain synchronized version of LinkedList, use the following utility method from Collections class.
   Public List synchronizedList(List);
7. Implements Serializable and Cloneable interface but not RandomAccess interface.
8. Best choice for frequent insertions and deletions in middle.
9. Worst choice for frequent retrieval operation.

Extra methods in LinkedList
In general, LinkedList can be used to implement Stack and Queue. To support this requirement LinkedList class contains 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();

Example:
```
import java.util.*;
class LinkedListDemo {
	public static void main(String[] args) {
		LinkedList l=new LinkedList();
		l.add("ten");
		l.add(10);
		l.add(null);
		l.addFirst("one");
		System.out.println(l); //[one, ten, 10, null]
		l.removeLast();
		l.removeFirst();
		System.out.println(l);//[ten, 10]
	}
}
```

Stack

It is a child class of Vector and contains only one constructor.

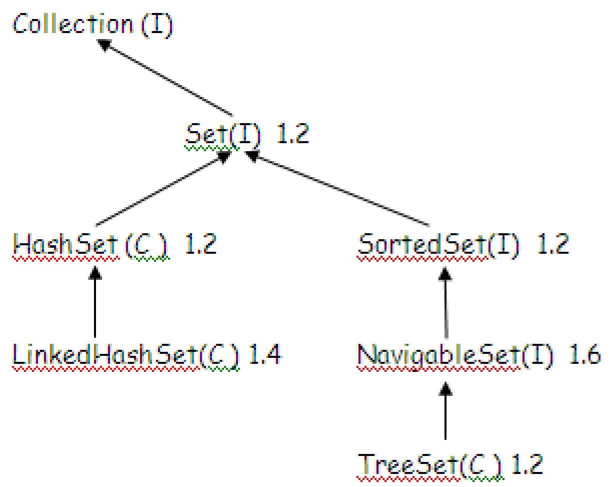
Stack s=new Stack();

Methods of Stack class:

1. Object push(Object o) → To insert an element at the top of the stack.
2. Object pop(); → To remove and return the top of the stack.
3. Object peek() → To return top of the stack without removal.
4. Boolean empty() → return true if the stack is empty.
5. Int search(Object o) → It returns distance from the top of the stack if the specified object exists, otherwise, return -1.

Example:
```
import java.util.*;
class StackDemo{
	public static void main(String[] args){
		Stack s=new Stack();
		s.push("A");
		s.push("B");
		s.push("C");
		System.out.println(s);//[A, B, C]
		System.out.println(s.search("A"));//3
		System.out.println(s.search("Z"));//-1
	}
}
```

Set <<interface>>



1. Set never allows duplicate elements.
2. The order of the Set elements depends on its implementation class.
    a. The elements in HashSet are unordered.
    b. The elements in LinkedHashSet are in insertion order.
    c. The elements in TreeSet are sorted (either Natural or Customized Sorting Order).
3. Set implementation classes allow atmost one null value(Zero or One null value).
4. By default, some of the Set implementation classes are sorted.
5. By default, none of the Set implementation class is synchronized (non thread-safe). Use following utility method from Collections class to make our set elements as thread-safe.

   Public static Set synchronizedSet(Set)
6. All Set implementation classes are Serializable and Cloneable but not RandomAccess.
7. The Set interface does not contain any new methods, hence we have to use Collection interface methods.
8. The HashSet uses hashing mechanism and LinkedHashSet uses both hashing and list mechanisms. To avoid duplicate elements, overriding both equals() and hashCode() methods is mandatory for HashSet and LinkedHashSet collections.
9. But, ThreeSet elements are sorted using either compareTo() method from Comparable interface or compare() method from Comparator interface. The return value of the compareTo() / compare() method is used to check duplicate elements in TreeSet. Hence, overrding equals() and hashCode() methods for TreeSet elements is not required.

Overriding equals() and hashCode() methods

To avoid duplicate elements in HashSet and LinkedHashSet, overrding equals() method is mandatory.
Since both HashSet and LinkedHashSet uses hashing mechanism, overrding hashCode() method is mandatory.

Example:

```
class Employee{
        @Override
        public int hashCode() {  }
        @Override
        public boolean equals(Object obj) {       }
}
```

HashSet

Internally uses hashing mechanism. Hence overrding hashCode() methods is mandatory.
Duplicate objects are not allowed. Hence overrding equals() method is mandatory.
HashSet is an unordered collection.
Allows atmost one null value.
We never sort HashSet elements i.e., explicitly also we cannot sort hashset elements.
By default, HashSet is not synchronized.

Constructors

1. HashSet()
   Create an empty HashSet object with default initial capacity 16, and default fill ratio is 0.75. Default fill ratio is also known as Load factor. The fill ratio is valid from **0 to 1**. After reaching fill ratio, JVM creates new hashset collection and copies all elements from old hashset to newly created hashset.
2. HashSet(int initial_capacity);
   Create an empty HashSet object with specified initial capacity, and default fill ratio is 0.75. It is always recommended to use higher initial capacity value.
3. HashSet(int initial_capacity, float fill_ratio); //fill_ratio in between 0 and 1 only
4. HashSet(Collection c);

Example#1:

```
import java.util.*;
class HashSetDemo{
        public static void main(String[] args){
                Set hs=new HashSet();
                hs.add("A");
                hs.add("B");
                hs.add("C");
                hs.add("C");
                hs.add(null);
                hs.add(null);
                System.out.println(hs.toString());
                System.out.println(h.size());//4
        }
}
```

O/P:

One of the possible output is: [null, A, B, C]

Example #2: Override both equals() and hashCode() methods for hashset elements.

```
class Employee{
        int eno;
        String ename;
        int sal;
        public Employee(int e, String name,int s ){
                eno = e;
                ename = name;
                sal = s;
        }
        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + ((ename == null) ? 0 : ename.hashCode());
                result = prime * result + eno;
                result = prime * result + sal;
                return result;
        }
        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                final Employee other = (Employee) obj;
                if (ename == null) {
                        if (other.ename != null)
                                return false;
                } else if (!ename.equals(other.ename))
                        return false;
                if (eno != other.eno)
                        return false;
                if (sal != other.sal)
                        return false;
                return true;
        }
}
public class HashSetDemo {
        public static void main(String[] args) {
                Employee e1 = new Employee(1,"abc", 1000);
                Employee e2 = new Employee(1,"abc", 1000);
                Set employees = new HashSet();
                employees.add(e1);
                employees.add(e2);
                System.out.println(employees.size()); //O/P: 1
        }
}
```

Note:

If we are trying to add duplicate object, we won't get any compiletime or runtime error instead add () method return **false**.

LinkedHashSet

This is exactly same as HashSet except the following.

HashSet	LinkedHashSet
HashSet uses Hashing mechanism to store and retrieve elements.	LinkedHashSet uses both hashing and list mechanisms to store and retrieve elements.
Unordered collection (Insertion order is not preserved).	Ordered collection based on insertion order but not index order(i.e., Insertion order is preserved).
Introduced in 1.2 version	Introduced in 1.4 version

Example:

```
import java.util.*;
class LinkedHashSetDemo{
        public static void main(String[] args){
                LinkedHashSet h=new LinkedHashSet();
                h.add(1);
                h.add(2);
                h.add(3);
                h.add(4);
                h.add(null);
                h.add(10);
                System.out.println(h);//[1, 2, 3, 4, null, 10]
        }
}
```

Note:

LinkedHashSet and LinkedHashMap are best suitable to implement caching applications because duplicate objects are not allowed and insertion order is preserved.

SortedSet <<interface>>

It is inherited from the Set interface.

The elements are sorted using java.lang.Comparable<interface> or using java.util.Comparator<interface>.

The SortedSet<<interface>> defines following 6 specific methods:

1. Public Object first();   //It returns first element of the SortedSet.
2. Public Object last(); //It returns last element of the SortedSet.
3. Public SortedSet headSet(Object obj); //It return the SortedSet whose elements are **less than** Object obj.
4. Public SortedSet tailSet(Object obj); //It returns the SortedSet whose elements are **greater than or equals to** Object obj.
5. Public SortedSet subSet(Object obj1, Object obj2);
   It returns the SortedSet whose elements are greater than or equals to obj1 and less than obj2.
6. Public Comparator comparator();
   It returns the underlying comparator or null in case of natural sorting order.

```
[101, 102, 103, 104, 105, 106, 107]
first()          --------------------  101
last()                 --------------------  107
headSet(105)     -------------------- [101, 102, 103, 104]
tailSet(105)           -------------------- [105, 106, 107]
subSet(102, 106) -------------------- [102, 103, 104, 105]
comparator()     --------------------  null
```

TreeSet

The TreeSet elements are in sorting order. Hence, the TreeSet elements should be Comparable or Comparator otherwise will throw **ClassCastException**.

The TreeSet internally uses sorting mechanism but not hashing mechanism hence overring hashCode() method is not required.

The return value of the compareTo() / compare() method is used to check duplicate elements, hence overrding equals() methods is not required.

Conclusion: In case of TreeSet, overrding both hashCode() and equals() methods are not required.

Constructors

1. TreeSet t= new TreeSet();
   The Treeset elements must be Comparable otherwise throws ClassCastException.
2. TreeSet t= new TreeSet(Comparator c);
   Creating an empty TreeSet object where the sorting order is customized sorting order.
3. TreeSet t= new TreeSet(Collection c);
   Make sure that collection must not contains null values, violation leands NPE.
4. TreeSet t= new TreeSet(SortedSet s);

Example:

```
import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;
class MyComparator1 implements Comparator {
        public int compare(Object o1, Object o2) {
                String s1 = (String) o1;
                String s2 = (String) o2;
                return -s1.compareTo(s2);
        }
}
public class TreeSetDemo1 {
        public static void main(String[] args) {
                SortedSet ss = new TreeSet(); // Natural sorting order using Comparable interface
                //SortedSet ss = new TreeSet(new MyComparator1()); //Customized sorting order.

                ss.add("A");
                ss.add("C");
                ss.add("B");
                ss.add("D");
                // ss.add(null); //NullPointerException
                ss.add("D");// duplicate elements are not allowed so D will be ignored,
                                    // no Compile time Error, no Runtime Exception.
                System.out.println(ss.toString());// [A, B, C, D] / [D, C, B, A]

                System.out.println(ss.first()); // A / D
                System.out.println(ss.last());  // D  / A
                System.out.println(ss.headSet("C")); //[A, B] / [d]
                System.out.println(ss.tailSet("C")); //[C, D] / [C, B, A]
        }
}
```

O/P: The output depends on which TreeSet constructor used:

New TreeSet()	New TreeSet(Comparator)
[A, B, C, D]	[D, C, B, A]

Question #1: Given the following code fragment

```
import java.util.TreeSet;
public class Test{
        public static void main(String... args) {
                TreeSet<Test> at = new TreeSet<Test>();
                at.add(new Test());
                at.add(new Test());
                at.add(new Test());
                System.out.println(at.size());
        }
}
```

What will be result of attempting to compile and run the given program?

a) Prints: “Test”
b) Prints: “0”
c) Prints: “-1”
d) Prints: “-1”
e) Prints nothing
f) Runtime exception
g) Compile time error

Ans) f

Note: Here TreeSet elements are not comparable hence throws ClassCastException.

<u>Null acceptance</u>

Case 1: We can insert null value as a first element to the empty TreeSet. After inserting null value, we cannot insert any other element, violation leands NPE.

Case 2: We cannot insert null values to the non-empty TreeSet, violation leands NPE.

Case 3: Make sure that collection must not contains null values before sorting them, violation throws NPE.

Example:

```
ArrayList al = new ArrayList();
Al.add(3);
Al.add(1);
Al.add(null);
SOP(al);   // [3, 1, null]
Collections.sort(al);  //NullPointerException
TreeSet ts = new TreeSet(al); //NullPointerException
```

NavigableSet<<interface>>

It is first time introduced in jdk1.6.

It defines several new methods to support navigation in TreeSet.

A NavigableSet may be accessed and traversed either in ascending or descending order.

New Methods:

1) headSet(Object, boolean inclusive)→ From the beginning till less than or equals to specified element (true means specified element is included).
2) tailSet(Object, boolean inclusive)→ From Specified element(false means specified element is excluded) till end.
3) Public Object pollFirst() → Removes the fisrt element
4) Public Object pollLast() → Removes the last element
5) public NavigabeSet descendingSet()
6) public Iterator descendingIterator()
7) lower(Object o)→Returns the value which is '**<**' specified value, or null if there is no such element.
8) higher(Object o)→ Returns the value which is **'>'**specified value, or null if there is no such element.
9) ceiling(Object o)→Returns the least element in this set **'>='** to the given element.
10) floor(Object o)→Returns the greatest element in this set **'<='** to the given element.

[2.0, 6.0, 7.0, 8.0, 10.0, 34.0]

higher(8.0) → 10.0
lower(8.0) → 7.0
Ceiling(7.5) → 8.0
Floor(7.5)→ 7.0
headSet(8.0)→ [2.0, 6.0 , 7.0]
headSet(8.0, true) → [2.0, 6.0 , 7.0, 8.0]
tailSet(8.0) → [8.0,10.0, 34.0]
tailSet(8.0, false) → [10.0, 34.0]

t.descendingSet() → [34.0, 10.0, 8.0, 7.0, 6.0, 2.0]
subSet(6.0,10.0)→ [6.0, 7.0, 8.0]
pollFirst() →[6.0, 7.0, 8.0, 10.0, 34.0]
pollLast()→[6.0, 7.0, 8.0, 10.0]

Example #1:

```
import java.util.Comparator;
import java.util.TreeSet;
import java.util.*;
class MyComparator2 implements Comparator {
	public int compare(Object o1, Object o2) {
		String s1 = (String) o1;
		String s2 = (String) o2;
		return -s1.compareTo(s2);
	}
}
Public class TreeSetDemo2 {
	public static void main(String[] args) {
		NavigableSet ns1 = new TreeSet(); // Natural sorting order using Comparable interface
		ns1.add("A");
		ns1.add("C");
		ns1.add("B");
		ns1.add("D");
		System.out.println(ns1);// [A, B, C, D]

		System.out.println(ns1.headSet("C")); //[A, B]
		System.out.println(ns1.headSet("C", true)); //[A, B, C]

		System.out.println(ns1.tailSet("C")); //[C, D]
		System.out.println(ns1.tailSet("C", false)); //[D]

		NavigableSet ns2 = ns1.descendingSet();
		System.out.println(ns1.toString()); // [A, B, C, D]
		System.out.println(ns2.toString()); // [D, C, B, A]

		System.out.println(ns1.pollFirst()); // A
		System.out.println(ns2.pollLast());  // B

		System.out.println(ns1.toString()); //[C, D]
		System.out.println(ns2.toString()); //[D, C]
	}
}
```

Example #2:

```
import java.util.ArrayList;
import java.util.List;
import java.util.NavigableSet;
import java.util.TreeSet;
public class NavigableSetDemo{
	public static void main(String[] args){
		TreeSet t = new TreeSet();
		t.add(34.0);
		t.add(6.0);
		t.add(2.0);
```

```
                t.add(8.0);
                t.add(7.0);
                t.add(10.0);
                System.out.println(t); // [2.0, 6.0, 7.0, 8.0, 10.0, 34.0]

                System.out.println(t.higher(8.0));  //10.0
                System.out.println(t.lower(8.0));  //7.0
                System.out.println(t.ceiling(7.5)); //8.0
                System.out.println(t.ceiling(34.5)); //null
                System.out.println(t.floor(7.5)); //7.0
                System.out.println(t.floor(1.5));//null
                System.out.println(t.headSet(8.0));   //[2.0, 6.0, 7.0]
                System.out.println(t.headSet(8.0,true));  //[2.0, 6.0, 7.0, 8.0]
                System.out.println(t.tailSet(8.0));    //[8.0, 10.0, 34.0]
                System.out.println(t.tailSet(8.0,false)); //[10.0, 34.0]

                NavigableSet t1 = t.descendingSet();
                System.out.println(t1);  //[34.0, 10.0, 8.0, 7.0, 6.0, 2.0]

                System.out.println(t.subSet(6.0,10.0)); //[6.0, 7.0, 8.0]
                t.pollFirst();
                System.out.println(t); //[6.0, 7.0, 8.0, 10.0, 34.0]
                t.pollLast();
                System.out.println(t); //[6.0, 7.0, 8.0, 10.0]
        }
}
```

Question#1: What is the output for the below code?

```
public class Test{
        public static void main(String... args){
                List  lst = new ArrayList();
                lst.add(34);
                lst.add(6);
                lst.add(6);
                lst.add(6);
                lst.add(6);
                lst.add(5);
                NavigableSet  nvset = new TreeSet(lst); //[5,6,34]
                System.out.println(nvset.tailSet(6));
        }
}
```

Select one from the below options:

A)Compile error : No method name like tailSet()

B)6 34

C)5

D)5 6 34

Answer: **B**

Note: The tailSet(6) Returns elements which are greater than or equal to 6.

Question#2: What is the output for the below code ?

```
import java.util.ArrayList;
import java.util.List;
import java.util.NavigableSet;
import java.util.TreeSet;
public class Test {
            public static void main(String... args) {
                        List lst = new ArrayList();
                         lst.add(34);
                         lst.add(6);
                         lst.add(6);
                         lst.add(6);
                         lst.add(6);

                 NavigableSet<Integer>  nvset = new TreeSet(lst); //[6,34]
                 nvset.pollFirst();
                 nvset.pollLast();
                 System.out.println(nvset.size());
            }
}
```

Select one of the following options:
A)Compile error : No method name like pollFirst() or pollLast()
B)0
C)3
D)5
Answer : B

Comparison among Set implementation classes.

Property	**HashSet**	**LinkedHashSet**	**TreeSet**
Data Structure	Hashing Mechanishm	Both Hashing + list mechanisms.	Balanced Tree (sorting mechanism)
Elements order	Un ordered	Insertion order	Sorted order
Duplicate objects	Not allowed	Not allowed	Not allowed
Heterogeneous Objects	Allowed	Allowed	Heterogeneous elements are not allowed in case of natural sorting order. But, incase of customized sorting order we can use heterogeneous elements.
Null acceptance	Allowed	Allowed	Allowed as a first element to the empty TreeSet otherwise **NPE**

Queue <<interface>>

The collection properties such as duplicate, ordered, null values and sorted depends on its implementation classes:

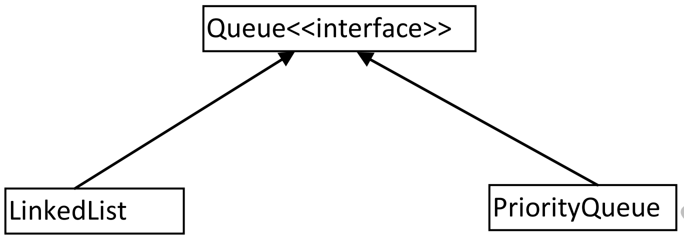
Property	LinkedList	PriorityQueue
Duplicate	Allows duplicate elements	Never allows duplicate elements.
Ordered	FIFO (Insertion order)	Sorting order
Null values	Allows duplicate null values.	Never allows null value at all.
Sorting	No sorting order rather only insertion order.	Either natural or customized sorting order.
Synchronized	No	No

The order of queue elements depends on its implementation classes:

a) In case of **LinkedList**, the order of the queue elements is First In First Out (FIFO) manner. Hence insertion order is preserved.

```
List list = new LinkedList();      //index order
Queue q = new LinkedList();        //insertion order
```

b) In case of **PriorityQueue**, the order of the queue elements depends on supplied comparator or natural sorting order. Hence insertion order is not preserved rather sorting order is preserved.



FIFO Order

Elements order depends on supplied comparator or natural sorting order.

Methods

	Throws Exception	**Return null or false**
Insert	Add(e)	Offer(e)
Remove	Remove()	Poll()
Examine	Element()	Peek()

The remove() and poll() methods remove and return head of the queue. Exactly which element is removed depends on the queue's ordering policy, which differs from implementation to implementation.
The element() and peek() methods return, but do not remove, head of the queue. Exactly which element is returned depends on the queue's ordering policy, which differs from implementation to implementation.
The methods remove() and element() throws **NoSuchElementException** in case of empty queue.
But, The methods poll() and peek() returns **null** in case of empty queue.

LinkedList (FIFO)

```
[b, a, c, e, d]
Offer("z")     →     [b, a, c, e, d, z]
add("y")       →     [b, a, c, e, d, z, y]
element()      →     [b, a, c, e, d, z, y]
peek()         →     [b, a, c, e, d, z, y]
poll()         →     [a, c, e, d, z, y]
remove()       →     [c, e, d, z, y]
```

Example #1:

```
import java.util.LinkedList;
import java.util.Queue;
public class QueueDemo1 {
   public static void main(String[] args) {
     Queue qe=new LinkedList(); //Elements order is insertion order i.e., FIFO
     qe.add("b");
     qe.add("a");
```

```
        qe.add("c");
        qe.add("e");
        qe.add("d");
        System.out.println(qe.toString()); //[b, a, c, e, d]

        qe.remove();
        qe.poll();
        System.out.println(qe.toString());//[c, e, d]

        qe.element();
        qe.peek();
        System.out.println(qe.toString()); //[c, e, d]

        qe.offer("z");
        qe.add("y");
        System.out.println(qe.toString());//[c, e, d, z, y]
    }
}
```

PriorityQueue
The order of the elements depends on which PriorityQueue() constructor used:

Public PriorityQueue() → Natural sorting order
Public PriorityQueue(Comparator)→ Customized sorting order

We cannot get the expected results while we iterate through priority queue. The iterator does not necessarilly go through the elements in the order of their Priority. In other words, if we want to see how the elements are added and removed, do **not** rely on iterator.
Null insertion is not possible even as the first element, otherwise throws NullPointerException.

Example:

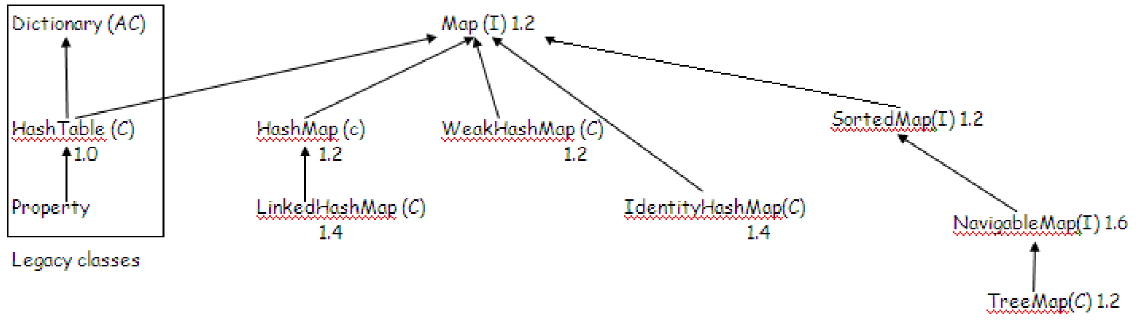
```
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Comparator;

class MyComparator implements Comparator<String>{
        public int compare(String str1, String str2) {
                return -str1.compareTo(str2);
        }
}
public class QueueDemo2 {
        public static void main(String[] args) {
                Queue q = new PriorityQueue(); // uses natural sorting order.
                //Queue q = new PriorityQueue(new MyComparator()); // uses customized sorting order.

                q.add("b");
                q.add("d");
                q.add("z");
                q.add("c");
                System.out.println(q.toString()); //[b, c, z, d]

                System.out.println(q.remove());//b
                System.out.println(q.remove()); //c

                System.out.println(q.remove());//d
                System.out.println(q.remove());//z
        }
}
```

O/P: The output depends on which PriorityQueue constructor used:

New PriorityQueue()	**New PriorityQueue(capacity,Comparator)**
b c d z	z d c b

Map <<Interface>>

If we want to represents a group of objects as a key and value pairs then we should go for Map i.e., maps key with value.
Both keys and values must be objects. We can use any class as key but which must override both hashCode() and equals() methods.
Keys must not be duplicated, but values can be duplicated.
Some of the Map implementation classes are ordered based on key.
Some of the Map implementation classes allows both null key & null values.
Some of the Map implementation classes are Sorted based on key either in natural or customized sorting order.
By default, some of the Map implementation classes are synchronized.
Each key and value pair is called as one entry. Hence a map is considered as a group of entries.



Key	**Value**
“A”	Apple
“B”	Ball
“C”	Cat
“D”	Dog

Methods

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

To insert one Key & value pair in the Map. If the specified key is already present then old value is replaced with new value and old value will be returned. If the key is not present, then null will be returned.

2. **Void putAll(Map m1);**
   Insert the group of entries from the specified map.
3. **Object get(Object key);**
   Return the value associated with the key. If the specified key is not available then it returns null.
4. **Object remove(Object key);**
   To remove the key and value entry and return value or null if the key does not exist.
5. **Boolean containsKey(Object key);**
   Check whether the key is present or not.
6. **Boolean containsValue(Object value);**
   Check whether the value is present or not.
7. **Boolean isEmpty();**
8. **Void clear();**
   Remove all entries.
9. **Int size();**
   Number of entries in the Map.
10. **Set keySet();** Returns the Set view of the keys, since keys cannot be duplicated.
11. **Collection values();** Returns Collection view of the values

All Map subclasses except TreeMap, the classes used as keys must override both hashCode() and equals() methods.

```
class Employee2 {
        @Override
        public int hashCode() {   }

        @Override
        public boolean equals(Object obj) {   }
}
```

Hashtable

Underlying data structure is hashing mechanism.
Keys cannot be duplicated but values can be duplicated.
Both keys and values are unordered i.e., the entries are unordered.
Neither key nor value can be null.
The hashtable entries are not sorted.
By default, Hashtable is a synchronized collection.

Two parameters which affects the hashtable performance are Capacity and Load factor. The capacity is the number of buckets in the hash table. The load factor is a measure of when hash table is automatically increased. The default load factor is 0.75.

The following methods are used to retrieve keys or values from the hashtable.

```
                Enumeration keys();
                                        } //Keys
```

```
        Set keySet();

        Enumeration elements() ;  }
        Collection values();      }  //values
```

Example #1: Use built-in class String as Key.

```
import java.util.Hashtable;
public class HashtableDemo1 {
    public static void main(String[] args) {
        Hashtable ht = new Hashtable();
        ht.put("one", 1);
        ht.put("two", 2);
        ht.put("three", 3);
        //ht.put("four", null); //NullPointerException
        //ht.put(null, 4);      //NullPointerException
        System.out.println(ht);  //{two=2, one=1, three=3}

        //Retrieve values from hashtable.
        Enumeration values = ht.elements();
        //Collection values = ht.values();
        while(values.hasMoreElements()){
            Integer iRef = (Integer)values.nextElement();
            System.out.print(iRef + "\t");
        }
        System.out.println();

        Enumeration keys = ht.keys();
        //Set keys = ht.keySet();
        while(keys.hasMoreElements()){
            String str = (String)keys.nextElement();
            System.out.print(str+"\t");
        }
    }
}
```

Example #2: Use User defined class as Key.

```
import java.util.Enumeration;
import java.util.Hashtable;
class Account{
    private int ano;
    private String aname;

    public Account(int ano, String aname) {
        super();
        this.ano = ano;
```

```
                this.aname = aname;
        }

        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + ((aname == null) ? 0 : aname.hashCode());
                result = prime * result + ano;
                return result;
        }

        @Override
        public boolean equals(Object obj) {
                if (this == obj)
                        return true;
                if (obj == null)
                        return false;
                if (getClass() != obj.getClass())
                        return false;
                Account other = (Account) obj;
                if (aname == null) {
                        if (other.aname != null)
                                return false;
                } else if (!aname.equals(other.aname))
                        return false;
                if (ano != other.ano)
                        return false;
                return true;
        }

        @Override
        public String toString(){
                return ano+" "+aname;
        }
}
public class HashtableDemo2 {
        public static void main(String[] args) {
                Account a1 = new Account(1, "aspire");
                Account a2 = new Account(2, "ramesh") ;
                Account a3 = new Account(2, "ramesh") ; //duplicate

                Hashtable ht = new Hashtable();

                ht.put(a1, "current");
```

```
		ht.put(a2, "savings");
		ht.put(a3, "savings");
		System.out.println(ht.toString());

		// Read keys
		// Set keys = ht.keySet();
		Enumeration keys = ht.keys();
		while (keys.hasMoreElements()) {
			Account key = (Account) keys.nextElement();
			System.out.print(key + "\t");
		}
		System.out.println();
		// Read values
		//Collection values = ht.values();
		Enumeration values = ht.elements();
		while (values.hasMoreElements()) {
			String value = (String) values.nextElement();
			System.out.print(value + "\t");
		}
	}
}
```

O/P:

```
{2 ramesh=savings, 1 aspire=current}
2 ramesh	1 aspire
savings	current
```

HashMap

The underlying data structure is hashing mechanism.
Keys cannot be duplicated but values can be duplicated.
Allows both null key and null values.
The entries are unordered.
The HashMap entries are not sorted.
By default, HashMap is not synchronized. But, we can explicitly synchronize HashMap by using following utility method from Collections class:

Public static Map synchronizedMap(Map m);

<u>**Example:**</u>

```
import java.util.HashMap;
import java.util.Iterator;
public class HashMapDemo {
	public static void main(String[] args) {
		HashMap ht = new HashMap();
		ht.put("driverClassName", "oracle.jdbc.driver.OracleDriver");
		ht.put("url", "jdbc:oracle:thin:@localhost:1521:xe");
		ht.put("username", "system");
		ht.put("password", "manager");
		ht.put(null, null);
		System.out.println(ht.toString());

		// Read keys
		Iterator keys = ht.keySet().iterator();
		while (keys.hasNext()) {
			String key = (String) keys.next();
			System.out.print(key + "\t");
		}
		System.out.println();

		// Read values
		Iterator values = ht.values().iterator();
		while (values.hasNext()) {
			String value = (String) values.next();
			System.out.print(value + "\t");
		}
	}
}
```

Difference between HashMap and Hashtable

HashMap	Hashtable
By default, it is not synchronized (non thread-safe)	By default, it is synchronized (thread-safe).
Performance is high	Performance is low
Introduced in 1.2 version	Legacy, introduced in 1.0 version.
Both key and values can be null.	Neither key nor value is null.

LinkedHashMap

This is exactly same as HashMap except the following differences:

HashMap	LinkedHashMap
The underlying data structure is hashing mechanism.	The underlying data structure is both hashing and list mechanisms.
The entries are unordered.	Entries are in insertion order based on keys.
Introduced in 1.2 version	Introduced in 1.4 version

Example:

```
import java.util.LinkedHashMap;
public class LinkedHashMapDemo {
	public static void main(String[] args) {
		LinkedHashMap ht = new LinkedHashMap();
		ht.put("driverClassName", "oracle.jdbc.driver.OracleDriver");
		ht.put("url", "jdbc:oracle:thin:@localhost:1521:xe");
		ht.put("username", "system");
		ht.put("password", "manager");
		ht.put(null, null);
		System.out.println(ht.toString());
	}
}
```

Note:
The LinkedHashSet and LinkedHashMap are best suitable for implementing caching application.

IdentityHashMap

This is exactly same as HashMap except the following differences:

HashMap	IdentityHashMap
The JVM always uses **equals()** to check duplicate keys i.e., content comparison.	The JVM uses "==" operator to check duplicate keys, i.e., reference comparison.

Example:

```
import java.util.IdentityHashMap;
import java.util.Map;
class IdentityHashMapDemo {
	public static void main(String[] args) {
		//Map<Integer, String> m = new HashMap<Integer, String>();
		Map<Integer, String> m = new IdentityHashMap<Integer, String>();
		Integer I1 = new Integer(10);
		Integer I2 = new Integer(10);
		m.put(I1, "aspire");
		m.put(I2, "technologies");
		System.out.println(m);
	}
}
```

O/P: The output depends on which HashMap or IdentityHashMap used:

HashMap	**IdentityHashMap**
{10=technologies} **Note:** I1.equals(I2) returns true, hence old value is replaced with new value.	{10=aspire, 10=technologies} **Note:** I1 == I2 returns false, hence two entries are entered into map.

WeakHashMap

This is exactly same as HashMap except the following differences.

HashMap	WeakHashMap
The object is not eligible for garbage collection even though it doesn't have any external references when it is associated with HashMap. i.e., HashMap dominates Garbage Collection.	If an object doesn't have any external references then it is always eligible for Garbage Collection, even though it is associated with WeakHashMap. i.e. Garbage Collection dominates WeakHashMap.

Example:

```
import java.util.*;
class Temp{
	public String toString(){ return "temp"; }
	protected void finalize(){
		System.out.println("finalize method");
	}
}

public class WeakHashMapDemo {
        public static void main(String[] args) throws Exception {
                //Map<Temp, String> m=new HashMap<Temp, String>();
                Map<Temp, String> m = new WeakHashMap<Temp, String>();
                Temp t = new Temp();
                m.put(t, "aspire");
                System.out.println(m);
                t = null;
                System.gc();
                Thread.sleep(5000);
                System.out.println(m);
        }
}
```

Output:

Map m = new HashMap()	Map m = new WeakHashMap()

{temp=aspire}
{temp=aspire}
	{temp=aspire}
	finalize method
	{}

SortedMap <<interface>>

If we want to insert a set of entries according to natural or customized sorting order of keys then we should go for SortedMap interface.

SortedMap interface defines the following six specific methods.

1. Object firstKey();
2. Object lastKey();
3. SortedMap headMap(Object key);   //key is exclusive
4. SortedMap tailMap(Object key);   //key is inclusive.
5. SortedMap subMap(Object key1, Object key2); //key1 is inclusive, but key2 is exclusive.
6. Comparator comparator();

TreeMap

TreeMap uses sorting mechanism but not hashing mechanism. Hence overrding hashCode() method is not required.
TreeMap never allows duplicate keys, but values can be duplicated. The duplicate keys are identified with the return value of the compareTo() / compare() methods, hence overriding equals() is not required.
The TreeMap entries are in sorting order based on keys.
Null Acceptance:

1. For empty TreeMap, the first entry would be with null key. But we should not insert anymore keys, violation leads NPE.
2. For non-empty TreeMap, we never insert entries with null key, violation leads NPE.

By default, the TreeMap entries are sorted based on key either in Natural or Customized sorting order.
By default, the TreeMap entries are not Synchronized. To synchronize explicitly, use the following utility method from Collections class:

Public Map synchronizedMap(Map);

Example:

```
import java.util.Comparator;
import java.util.SortedMap;
import java.util.TreeMap;
class MyComparator3 implements Comparator{
        public int compare(Object o1, Object o2) {
```

```
                    Integer i1 = (Integer)o1;
                    Integer i2 = (Integer)o2;
                    return -i1.compareTo(i2);
          }
}
public class TreeMapDemo1{
          public static void main(String[] args){
                    SortedMap sm=new TreeMap();  //Natural sorting order.
                    //SortedMap sm=new TreeMap(new MyComparator3());  //customized sorting order.
                    sm.put(103,"abc");
                    sm.put(100,"xyz");
                    sm.put(104,"mnr");
                    sm.put(101,"pqr");
                    //sm.put("FFF","aaa");//java.lang.ClassCastException
                    //sm.put(null,"aaa");//java.lang.NullPointerException
                    System.out.println(sm); //{100=xyz, 101=pqr, 103=abc, 104=mnr}

                    System.out.println(sm.headMap(103)); //{100=xyz, 101=pqr}
                    System.out.println(sm.tailMap(103)); //{103=abc, 104=mnr}
          }
}
```

O/P: The output depends on which TreeMap constructor used.

TreeMap()	**TreeMap(new MyComparator2())**
{100=xyz, 101=pqr, 103=abc, 104=mnr}	{104=mnr, 103=abc, 101=pqr, 100=xyz}

<u>NavigableMap</u>
It is inherited from SortedMap.
This interface defines the following methods to support Navigation.
<u>Methods</u>
1. headMap(String key, boolean inclusive) // From the beginning till less than or equals to specified element (true means specified element is included).
2. tailMap(String key, boolean inclusive) // From Specified element(false means specified element is excluded) till end.
3. pollFirstEntery(); → Removes first entry.
4. pollLastEntry();→ Removes last entry.
5. Public NavigableMap descendingMap();→ Returns the reverse order of the key.
6. Public NavigableSet decendingKeySet()
7. ceilingKey(e); → Returns the least key in this map greater than or equal to the given key.
8. floorKey(e); → Returns the greatest key in this map less than or equal to the given key.
9. HigherKey(e);→ Returns the key which is greater than specified key, or null if there is no such key.
10. lowerKey(e); → Returns the key which is less than specified key, or null if there is no such key.

{A=Apple, B=Ball, C=Cat, D=Dog}
ceilingKey(C) → C
flooringKey(C) → C
higherKey(C) → D
lowerKey(C) → B
pollFirstEntry() → { B=Ball, C=Cat, D=Dog}
pollLastEntry() → { B=Balls, C=Cat }
descendingMap() →{C=Cat, B=Ball}

Example:

```
import java.util.Comparator;
import java.util.NavigableMap;
import java.util.TreeMap;
class MyComparator4 implements Comparator{
	public int compare(Object o1, Object o2) {
		Integer i1 = (Integer)o1;
		Integer i2 = (Integer)o2;
		return -i1.compareTo(i2);
	}
}
public class TreeMapDemo2{
	public static void main(String[] args){
		NavigableMap nm=new TreeMap();  //Natural sorting order.
		//NavigableMap nm=new TreeMap(new MyComparator4());  //customized sorting order.
		nm.put(103,"abc");
		nm.put(100,"xyz");
		nm.put(104,"mnr");
		nm.put(101,"pqr");
		//nm.put("FFF","aaa");//java.lang.ClassCastException
		//nm.put(null,"aaa");//java.lang.NullPointerException
		System.out.println(nm); //{100=xyz, 101=pqr, 103=abc, 104=mnr}

		System.out.println(nm.headMap(103)); //{100=xyz, 101=pqr}
		System.out.println(nm.headMap(103, true)); //{100=xyz, 101=pqr, 103=abc}

		System.out.println(nm.tailMap(103)); //{103=abc, 104=mnr}
		System.out.println(nm.tailMap(103, false)); //{104=mnr}
	}
}
```

O/P: The output depends on which TreeMap() constructor used:

New TreeMap()	New ThreeMap(new MyComparator3())
{100=xyz, 101=pqr, 103=abc, 104=mnr}	{104=mnr, 103=abc, 101=pqr, 100=xyz}

Properties

Application properties and Server or System properties are usually defined in property file with .properties extension.

The entries in property file are key & values pairs with separater equal (=) symbol.

Though the Properties class is inherited from Hashtable but both keys and values must be always strings.

The advantages of using properties file is, the changes can be made in one place, which are reflected across all other placess in our code.

Important methods of Properties Class:

1. String getProperty(String Propertyname);
   Return the value associated with the specified property or null if the property does not exist.
2. String setProperty(String popertyname, String propertyvalue);
3. Enumeration propertyNames();
4. Set stringPropertyNames();
5. Void load(InputStream is);

To load the properties from properties file into java properties object.

6. Void Store(OutputStream os, String comment);
   To store the properties from java properties objects to the properties file.

Example:

#connection.properties

```
driverClassName=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:xe
username=system
password=manager
```

//PropertiesDemo.java

```
import java.io.FileInputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
public class PropertiesDemo {
	private static Properties props = null;
	static{
		try{
			props = new Properties();
			props.load(new FileInputStream("connection.properties"));
		}catch(Exception e){
			e.printStackTrace();
		}
	}

	public static Connection getConnection(){
		Connection con = null;
		String driverClass = props.getProperty("driverClassName");
		String url = props.getProperty("url");
		String username = props.getProperty("username");
		String pwd = props.getProperty("password");
		try{
			//Load & Register JDBC Driver class
			Class.forName(driverClass);

			con = DriverManager.getConnection(url, username, pwd);
		}catch(SQLException e){
			e.printStackTrace();
		}catch (Exception e) {
			e.printStackTrace();
		}
		return con;
	}

	public static void main(String[] args) {
		Connection con = PropertiesDemo.getConnection();
		System.out.println(con);
	}
```

}

Collections <<class>>

This class present in the java.util package and define several utility methods for collection implemented class objects.

a) Sorting a List

Collections class contains the following methods to perform the sorting of the list implemented class objects.

1. Public static void sort(List l);
   Sorts list elements in natural sorting order. In this case the objects should be Homogeneous and comparable otherwise we will get ClassCastException.
2. Public static void sort(List l, Comparator c);
   Sorts list elements in customized sorting orde described by comparator objects. In this case the objects can be non homogeneous and non-comparable.

Make sure that list elements must not contains null elements before sorting,violation leads NullPointerException.

Example:

```
import java.util.*;
import java.util.Collections;
class MyComparator3 implements Comparator{
        public int compare(Object Obj1, Object Obj2){
                String s1=(String)Obj1;
                String s2=(String)Obj2;
                return s2.compareTo(s1); //descending order
        }
}
class CollectionClassDemo{
        public static void main(String[] args){
                ArrayList l=new ArrayList();
                l.add("Z");
                l.add("A");
                l.add("K");
                l.add("N");
                //l.add(new Integer(10)); //ClassCastException
                //l.add(null); // NullPointerException
                //Collections.sort(l); //Natural sorting order.
                Collections.sort(l, new MyComparator3()); //customized sorting order.
                System.out.println(l);
        }
}
```

O/P: The output depends on which overloaded sort() method used:

Collections.sort(list) //natural sorting order	Collections.sort(list, new MyComparator3());
[A, K, N, Z]	[Z, N, K, A]

b) Searching an Object in List

Before calling search method, list elements must be explicitly sorted either in natural or customized sorting order, otherwise, we will get **un-predictable** results.

The Collections class contains the following search methods:

1. Public static int binarySearch(List l, Object obj);
   Before searching, list elements must be sorted using natural sorting order.
2. Public static int binarySearch(List l, Object obj, Comparator c);
   Before searching, list elements must be sorted using customized sorting order, and same comparator must be used.

Conclusions

1. Internally this search method uses Binary Search algorithm.
2. For successful search, it returns index.
3. For un successful search, it return insertion point.
4. Insertion point is the place where we can place the object without disturbing sorting order.
5. Ensure that the list should be sorted before calling search method otherwise we will get un-predictable results.
6. If the list is sorted according to the comparator then at the time of search operation we should also pass same comparator object otherwise we will get un-predictable results.

Example:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
class MyComparator5 implements Comparator{
        public int compare(Object o1, Object o2) {
                Integer i1 = (Integer)o1;
                Integer i2 = (Integer)o2;
                return -i1.compareTo(i2);
        }
}
public class BinarySearchDemo {
        public static void main(String[] args) {
                List aList = new ArrayList();
                aList.add(7);
                aList.add(1);
                aList.add(3);
                aList.add(5);

                Collections.sort(aList); //natural sorting order.
                System.out.println(aList); //[1, 3, 5, 7]
                System.out.println(Collections.binarySearch(aList,7)); //3
                System.out.println(Collections.binarySearch(aList,1)); //0
                System.out.println(Collections.binarySearch(aList,4)); //-3
                System.out.println(Collections.binarySearch(aList,8)); //-5
                System.out.println(Collections.binarySearch(aList,0)); //-1

                Collections.sort(aList, new MyComparator5());//customized sorting order.
                System.out.println(aList); //[7, 5, 3, 1]
                System.out.println(Collections.binarySearch(aList,5, new MyComparator5())); //1
                System.out.println(Collections.binarySearch(aList,1, new MyComparator5()));//3
                System.out.println(Collections.binarySearch(aList,3, new MyComparator5())); //2
                //System.out.println(Collections.binarySearch(aList,3)); // Un-predictable results.

        }
}
```

Given a properly prepared collection containing **five** elements, the range of results could a proper invocation of Collections.binarySearch() produce is **-6 through 4**.

Example:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
public class BinarySearchRange {
        public static void main(String[] args) {
                List aList = new ArrayList();
                aList.add(1);
                aList.add(3);
                aList.add(5);
                aList.add(7);
                aList.add(9);

                Collections.sort(aList); //natural sorting order.
                System.out.println(aList); //[1, 3, 5, 7, 9]
                System.out.println(Collections.binarySearch(aList,1)); //0
                System.out.println(Collections.binarySearch(aList,9)); //4
                System.out.println(Collections.binarySearch(aList,0)); //-1
                System.out.println(Collections.binarySearch(aList,10)); //-6
        }
}
```

The range is: **-6 through 4**

Given a properly prepared collection containing **'n'** elements, the range of results could a proper invocation of Collections.binarySearch() produce is: -**(n+1) through +(n-1) results.**

c) <u>Reversing list elements</u>

The Collections class contains following method to perform reverse operation:

Public static void **reverse**(List l);

<u>Example:</u>

```
import java.util.ArrayList;
import java.util.Collections;
public class ReverseOrder {
        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);
                l.add(new Integer(10));
                l.add(null);
                System.out.println(l);//   [15, 0, 20, 10, 5, ten, null]
                Collections.reverse(l);
                System.out.println(l);//   [null, ten, 5, 10, 20, 0, 15]
        }
}
```

d) Reverse Order

Collections class have overloaded reverseOrder() method to return Comparator which is reverse of the given sorting order.

I. Public static **Comparator reverseOrder()** //Returns the Comparator which is reverse of the natural sorting order.

Example:

```
ArrayList aList = new ArrayList();
aList.add("1");   aList.add("2");   aList.add("3");   aList.add("4");   aList.add("5");
aList.add("6");   aList.add("7");   aList.add("8");
Comparator c = Collections.reverseOrder(); //Return type is Comparator
Collections.sort(aList, c);
Int result = Collections.binarySearch(aList, "6", c);  //Returns : 2
```

II. Public static Comparator **reverseOrder(Comparator)** //Returns the Comparator which is reverse of the customized sorting order.

e) Synchronize collection elements

The following Collections class utility methods are used to synchronize collection elements explicitly.

```
Public static void List synchronizedList(List);
Public static void Set synchronizedSet(Set);
Public static void Map synchronizedMap(Map);
```

Arrays <<class>>

This class defines the several utility methods for primitive and Object arrays.

a) Sorting an Array

1. Public static void sort(primitive[] p);

   Primitive elements are always sorted according to the natural sorting order only.
2. Public static void sort(Object[] o);

   To sort according to natural sorting order
3. Public static void sort(Object[] o, Comparator c);

   To sort according to customized sorting order

Note:

We can sort object array either by natural or customized sorting order. But we can sort primitive arrays only by natural sorting order.

Example:

```
import java.util.Arrays;
import java.util.Comparator;
class MyComparator6 implements Comparator{
        public int compare(Object obj1, Object obj2){
                String s1=(String)obj1;
                String s2=(String)obj2;
                return s2.compareTo(s1);
        }
}
class ArraysSortingDemo{
        public static void main(String[] args) {
                int[] arr= {10, 5, 20, 11, 6};
                Arrays.sort(arr); //primitive types always uses natural sorting order.
                System.out.println("Primitive array After Sorting");
                for (int e: arr ){
                        System.out.print(e+"\t");
                }
                String[] sarr={"A", "Z", "C"};
                Arrays.sort(sarr); //natural sorting order.
                System.out.println("\nObject array After Sorting");
                for (String str:sarr ){
                        System.out.print(str+"\t");
                }
                Arrays.sort(sarr, new MyComparator6());//customized sorting order.
                System.out.println("\nObject array After Sorting By Comaparator");
                for (String str:sarr ){
                        System.out.print(str+"\t");
                }
        }
}
```

O/P:

```
Primitive array After Sorting
5       6       10      11      20
Object array After Sorting
A       C       Z
Object array After Sorting By Comaparator
Z       C       A
```

b) **Searching an Array**

Arrays class can define the following methods to perform search operations.

1. Public static int binarySearch(primitive[] p, primitive element);
2. Public static int binarySearch(Object[] o, Object element);

3. Public static int binarySearch(Object[] o, Object element, Comparator c);

Note:
All rules are exactly similar to Collections class binarySearch() method.

Example:

```
import java.util.Arrays;
import java.util.Comparator;
class ArraysSearchingDemo {
	public static void main(String[] args) {
		int[] arr = { 10, 5, 20, 11, 6 };
		Arrays.sort(arr);
		for(int a : arr){
			System.out.print(a+"\t");
		}
		System.out.println("\n"+Arrays.binarySearch(arr, 6));//1

		String[] sarr = { "A", "Z", "B" };
		Arrays.sort(sarr);
		for(String str : sarr){
			System.out.print(str+"\t");
		}
		System.out.println("\n"+Arrays.binarySearch(sarr, "Z"));//2
		System.out.println(Arrays.binarySearch(sarr, "S"));//-3

		Comparator c = Collections.reverseOrder();
		Arrays.sort(sarr, c);
		for(String str : sarr){
			System.out.print(str+"\t");
		}
		System.out.println("\n"+Arrays.binarySearch(sarr, "Z", c));//0
		System.out.println(Arrays.binarySearch(sarr, "S", c));//-2
	}
}
```

O/P:

```
5	6	10	11	20
1
A	B	Z
2
-3
Z	B	A
0
-2
```

c) asList() method

Arrays class contains asList() method to view existing array object in the list form.

Public static List asList(Object[] o);

After creating List reference, if we are performing any modifications on list, those changes are reflected in array, and vice versa.

By using List reference, we are not allowed to perform any operations which various the array size, violation leads **UnSupportedOperationException**.

Example:

```
import java.util.Arrays;
import java.util.*;
import java.util.Comparator;
class ArrayToListConversionDemo{
	public static void main(String[] args){
		String[] sarr={"A", "Z", "B"};
		List list=Arrays.asList(sarr);
		System.out.println(list);//[A, Z, B]
		sarr[0]="K";
		System.out.println(list);//[K, Z, B]
		list.set(1,"L");
		for (String str: sarr ){
			System.out.print(str+"\t");// K   L   B
		}
		//list.add("aspire");//UnsupportedOperationException
		//list.remove(2);//UnsupportedOperationException
	}
}
```

Generic types (type safe)

- **Introduction**
- **Mixing generic and non-generic code**
- **Type erasure**
- **Polymorphism with Generics**
- **Wildcard character (?)**
- **Generic classes (or Bounded classes)**
- **Generic methods**

Introduction

By default, arrays are type safe i.e., once we create an array of type employee, it can accept only employees. If we try to insert other than employees such as students, strings, etc, the compiler will generate an error.

Example:

```
Employee[] emps = new Employees[5];
Emps[0] = new Employee(); //Ok
Emps[1]=new Student(); //C.E   Incompatible types.  Found: Student, Required: Employee
Emps[2]="aspire" //C.E  Incompatible types.  Found: String, Required: Employee
```

An employee array always contains employees, hence, arrays are always type-safe.

By default, collections are not type safe, since we can add any type of object to the collection. For example, our requirement is only adding employees, but unknowingly we might be added students, but the compiler will not generate any compilation error.

Example:

```
ArrayList al = new ArrayList();
Al.add(new Employee());
Al.add(new Student()); //no compilation error. Non type safe
Al.add("aspire"); //no compilation error. Non type safe.
```

Limitations with non-generic (non type safe) collections:

1. There is ***No Guarantee*** for the element types in arraylist which contains only employees.
2. While reading array elements, type casting is not required. But in case of collections, ***Explicit type cast*** is required while retrieving elements even though all elements in collection are having same type.
   Example:
   ```
   ArrayList al = new ArrayList();
   Al.add(new Employee());
   Employee e = al.get(0); //C.E. Found: Employee, but Required: Object.
   Employee e = (Employee)al.get(0); //Ok.
   ```
3. While retrieving elements, there may be a chance of getting ***ClassCastException*** at runtime.
   Example:
   ```
   ArrayList al = new ArrayList();
   al.add("ten");
   al.add(10);
   String str = (String)al.get(0);//Ok
   String str = (String)al.get(1);//ClassCastException at runtime.
   ```

To resolve above problems, Generic types (type safe) were introduced in jdk1.5.
To add only string elements into ArrayList, we can create generic (type safe) version of ArrayList as follows:

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

For the above ArrayList, we can add only string elements. If we are trying to add other than string elements, we will get compilation error itself. Hence, Generics are type safe. Generic types are resolved at compile time.
Example:

Base type Generic or Parameter type

```
ArrayList<String> fruits  =  new ArrayList<String>(); //ArrayList holds only String objects.
fruits.add("apple");
fruits.add("grapes");
fruits.add("banana");
fruits.add(new Integer(10));//C.E. Cannot find symbol add(Integer)
```

Hence, it is guaranteed that all elements in generic collection are similar types.

While retrieving elements, we can directly assign to String variable without explicit type casting.

```
String fruit = fruits.get(0);
```

It is always recommended to use enhanced-for loop to retrieve elements from Generic collection as follows:

```
for(String fruit : fruits){
        SOP(fruit);
}
```

Generic type must be reference type (object type) but not primitive type, violation leads compile time error saying Unexpected type.

```
ArrayList<int> l= new ArrayList<int>(); // CE: unexcepted type found: int   required: reference type
```

Conversion (Refactor) from non generic code to generic code is simple:

List names = new ArrayList();	List<String> names = new ArrayList<String>();
Public List getNames(){}	Public List<String> getNames(){}
Public void setNames(List names){}	Public void setNames(List<String> names){}

Mixing Generic and Non-generic types

We can mix Generic and Non-Generic collections as follows:

1. List names = new ArrayList<String>(); // **Warning:** Reference type is a raw type, but it should be parameterized.
   The above declaration accepts any type of object.
   names.add("ten");//Ok
   names.add(10);//Ok.
   for(Object o : names){}//No compilation and runtime error.
2. List<String> names = new ArrayList();// **Warning**: Actual object type is raw type, but it should be parameterized.
   The above declaration accepts only string objects.
   names.add("ten"); //Ok
   //names.add(10); //Compilation error.
   for(String str : names){}//Ok
   for(Object o : names{})//Ok. But above for loop is recommended.
3. We should be careful while assigning non generic type object to generic type reference variable, there may be a chance of getting **ClassCastException**.
   Example:
   ```
   ArrayList al = new ArrayList();
   al.add("ten");
   al.add(10);
   List<String> names = al; //Warning but not compilation error.
   ```

```
        for(String name : names){     }//ClassCastException
        for(Object o : names){
                if(o instanceof String )
                        String str = (String)o;
                else if(o instanceof Integer)
                        Integer iRef = (Integer)o;
        }//Ok
```

But, below example will not throw ClassCastException.

```
        ArrayList al = new ArrayList();
        al.add("ten");
        al.add("10");
        List<String> names = al; //Warning but not compilation error.
        for(String name : names){     }
```

Note: The code that compiles with warnings is still successful compilation.

4. Also, we can assign generic type object to non generic type reference variable, but ClassCastException never thrown by the JVM.
   Example:

```
        List<String> names = new ArrayList<String>();
        names.add("abc");
        names.add("xyz");
        List list = names;
        //for(String str : list){}//C.E
        for(Object o : list){
                String str = (String)o; //No ClassCastException
        }
```

Type erasure

The generic type information is available only at compile time but not at runtime. The compiler will verify generic types and then removes the type information. Hence, none of the type information is available at runtime. At runtime, both non-generic and generic code looks exactly same.

Example:

```
        List<String> names = new ArrayList<String>();
```

After compilation:

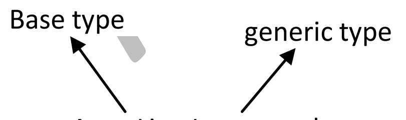
```
        List names = new ArrayList(); // type information is removed.
```

Hence generics are meant for compile time protection but not runtime protection since generic type information is not available at runtime.

Polymorphism with Generics

Polymorphism concept is possible only for base type but not for generic type:

Example:



```
        ArrayList<Integer> al = new ArrayList<Integer>();
        List<Integer> al = new ArrayList<Integer>(); //base type is polymorphic
        ArrayList<Object> al = new ArrayList<Integer>(); //C.E. Type mismatch: cannot convert from
                                                        ArrayList<Integer> to ArrayList<Object>
        ArrayList<Number> al = new ArrayList<Integer>(); //C.E. Type Mismatch
```

class Animal{}

class Dog extends Animal{}
class Cat extends Animal{}
We can pass any type of animal array (Dog[], Cat[], etc) to the method which takes Animal[] array. But we cannot add a different animal to array, violation leads to **ArrayStoreException.**
Example:

Void addAnimals(Animal[] animals){
    animals[0]=new Cat();//**throws ArrayStoreException**
}
Dog[] dogs = new Dog[3];
addAnimals(dogs);
	Void addAnimals(Animal[] animals){
	    animals[0]=new Dog();//**throws ArrayStoreException**
	}
	Cat[] cats = new Cat[3];
	addAnimals(cats);
		void addAnimals(Animal[] animals){
		    animals[0]=**new Dog();**
		    animals[1]=**new Cat();**
		}
		Animal[] animals = new Animal[10];
		addAnimals(animals);

We cannot pass any generic type of Animal (List<Dog>, List<Cat>, etc) to the method which takes Animal generic type. Violation leads compilation error.
Example:

Void addAnimals(List<**Animal**> animals){
}
List<Dog> dogs = new ArrayList<**Dog**>();
addAnimals(dogs);//C.E
	Void addAnimals(List<**Animal**> animals){ }
	List<Cat> cats = new ArrayList<**Cat**>();
	addAnimals(cats);//C.E
		Void addAnimals(List<Animal> animals){ }
		List<Animal> animals= new ArrayList<**Animal**>();
		addAnimals(animals); //Ok

Array type information is available at runtime. Since JVM knows array type, hence, it throws **ArrayStoreException** if we try to **add** different animal type.
But Generic type information is not available at runtime because of the type erasure. Since, JVM does not know generic type information, hence it cannot throw an exception if we are trying to add different animal type. Hence, the compiler itself generate an error if we are trying to assign different generic type of Animal to the method which takes Animal generic type.

Wildcard (?)

Wildcard (?) symbol supports generic type polymorphism.

class, abstract or interface
↑
1) **<? extends generic-type>**

The wildcard (?) symbol with 'extends' guarantees that , just we use collection for retrieving elements but not for adding new elements. We can send any generic type which is subtype of generic supertype.
For example, List<Animal>, List<Dog>, List<Cat> are all allowed.
Example:

```
import java.util.ArrayList;
import java.util.List;
public class WildcardDemo {
	public static void main(String[] args) {
		List<Animal> animals = new ArrayList<Animal>();
		animals.add(new Dog());
		animals.add(new Cat());
		readAnimals(animals); //Ok.

		List<Dog> dogs = new ArrayList<Dog>();
		dogs.add(new Dog());
		readAnimals(dogs); //Ok.

		List<Cat> cats = new ArrayList<Cat>();
		cats.add(new Cat());
		readAnimals(cats); //Ok.
	}
	private static void readAnimals(List<? extends Animal> animals){ //allows any subtype of Animal
		//animals.add(new Cat()); //C.E.
		//animals.add(new Dog()); //C.E.
		//Read elements
		for(Animal a : animals){
			System.out.println(a);
		}
	}
}
```

There is no <? **implements** Serializable> syntax. The keyword extends in the context of wildcard refers both implementation classes of an interface and subclasses.

Example:

Interface Animal{}
Class Dog implements Animal{}
void readAnimals(List<? **extends** Animal>){}
	abstract class Animal{}
	class Dog extends Animal{}
	void readAnimals(List <? extends Animal>){}

2) <? super generic-type>

This is used to **add** elements. Accept generic type which is same or any one of its super type, i.e., anything higher in the inheritance hierarchy, but not lower in the inheritance hierarchy.

For example, List<Dog>, List<Animal>, List<Object> are all allowed, but not List<Cat>.

Example:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class WildcardDemo1 {
	public static void main(String[] args) {
		List<Dog> dogs = new ArrayList<Dog>();
		dogs.add(new Dog());
		addDogs(dogs); //Ok.

		List<Animal> animals = new ArrayList<Animal>();
		animals.add(new Cat());
		addDogs(animals); //Ok

		List<Object> lists = new ArrayList<Object>();
		animals.add(new Cat());
		addAnimals(lists); //Ok

		/*List<Cat> cats = new ArrayList<Cat>();
		  cats.add(new Cat());
		  addDogs(cats); //Compilation error
		 */
	}
	private static void addDogs(List<? super Dog> dogs){
		dogs.add(new Dog());
		//dogs.add(new Animal()); //C.E. Not applicable for Animal
		//dogs.add(new Cat());//C.E. Not applicable for Cat
		/*Iterator ittr = dogs.iterator();
		while(ittr.hasNext())
			System.out.println(ittr.next());*/

		for(Object  o : dogs){  //always Object
			System.out.println(o);
		}
	}
}
```

3) <?>

Wildcard <?> without extends or super keywords means “any type”. This is exactly same as <? extends Object>, i.e., which accepts any type but for read only.
For example, List<Object>, List<Animal>, List<Dog>, List<Cat> are all allowed.
Difference between <?> and <Object>:

List<?>	List<Object>
Accepts any generic type.	Accepts only Object generic type.
Only for reading but not for adding.	Both reading and adding.

Wildcards can be used only for reference declarations. Wildcard notation cannot be used in object creation.
Example:
List<?> list = new ArrayList<Dog>();☑
List<? extends Animal> list = new ArrayList<Animal>();☑
List<? extends Animal> list = new ArrayList<Dog>();☑
List<? extends Animal> list = new ArrayList<Cat>();☑
List<? extends Animal> list = new ArrayList<Object>();☒
List<? super Dog> list = new ArrayList<Dog>();☑
List<? super Dog> list = new ArrayList<Animal>();☑
List<? super Dog> list = new ArrayList<Object>();☑
List<? super Animal> list = new ArrayList<Animal>();☑
List<? super Animal> list = new ArrayList<Object>();☑
List<? super Animal> list = new ArrayList<Dog>();☒
List<? super Animal> list = new ArrayList<Cat>();☒
List<? extends Animal> list = new ArrayList<**? extends Animal**>();☒

Bounded types (or Generic class)

The below example uses class parameter type i.e., the type declared with the class.

Non Generic class	**Generic class**

public class NonGenericClass{
    private **String** value;
    public String getValue() {
        return value;
    }
    public void setValue(String value){
        this.value = value;
    }
public static void main(String... args){
NonGenericClass obj = new NonGenericClass();
obj.setValue("ten");
//obj.setValue(10); //CE saying method
	public class GenericClass<**T**> {
	    private **T** value003B
	    public **T** getValue() {
	        return value;
	    }
	    public void setValue(**T** value){
	        this.value = value;
	    }
	public static void main(String... args){
	GenericClass<**Integer**> gc1 = new GenericClass<**Integer**>();
	gc1.setValue(10);
	System.*out*.println(gc1.getValue()); //10

setValue(String) is not applicable for setValue(int)
System.*out*.println(ngc.getValue());
}
}
	
	GenericClass<**String**> gc2 = new GenericClass<**String**>();
	gc2.setValue("ten");
	System.*out*.println(gc2.getValue()); //"ten"
	}
	}

We cannot use wildcard <?> with bounded classes and variables.

Example:

```
Public class NumberHolder<? extends Number>{} ☒// The <?> cannot be used with bounded classes.
Public class NumberHolder <?>{ ? aNum;}  ☒// The <?> cannot be used with variables.
Public class NumberHolder <T>{ T aNum;}  ☑
Public class NumberHolder <T extends Number>{ }  ☑
Public class NumberHolder <T implements Runnable>{} ☒ //implements cannot used with bounded types.
Public class NumberHolder <T super Integer>{ } ☒  //super cannot be used with bounded types.
```

Generic Method

We can individually declare any method as generic without declaring class as generic.

Example:

```
public class OnlyGenericMethod {
        //The Type <T> must be declared before return type.
        public <T> void setAnyValue(T t){
                System.out.println(t.getClass().getSimpleName());
        }
        public <T extends Number> void setAnyNumbers(T t){
                System.out.println(t.getClass().getSimpleName());
        }
        public static void main(String[] args) {
                OnlyGenericMethod m = new OnlyGenericMethod();
                m.setAnyValue(10);
                m.setAnyValue("10");

                m.setAnyNumbers(10);
                m.setAnyNumbers(10.0);
                //m.setNumbers("10"); //C.E. Type mismatch.
        }
}
```

Code Snippets:

1)

List<String> names = new LinkedList<String>();
names.add("aspire");
SOP(names.get(0).length());//Ok
	List<String> names = new LinkedList();
	names.add("aspire");
	SOP(names.get(0).length());//Ok
		List names = new LinkedList<String>();
		names.add("aspire");
		SOP(names.get(0).length());//C.E

2)

```
class MinMax<T extends Comparable<T>>{  //only comparable classes
        T min = null;
        T max = null;
        public MinMax(){}
        public void put(T value){}
        void meth(){
                MinMax<String> names = new MinMax<String>();
                MinMax<Integer> names1 = new MinMax<Integer>();
                //MinMax<StringBuffer> names2 = new MinMax<StringBuffer>();//Bound Mismatch
        }
}
```

3)

public void addStrings(List list) { list.add("foo"); list.add("bar"); } //compile with warnings.	public void addStrings(List<String> list) { list.add("foo"); list.add("bar"); }//compile without warnings.	public void addStrings(List<? **super** String> list) { list.add("foo"); list.add("bar"); }//compile without warnings.	public void addStrings(List<? **extends** String> list) { list.add("foo"); list.add("bar"); }//C.E

4) The following example shows the conversion from non-generic to generic method:

public static int sum(List list) {
  int sum = 0;
  for (Iterator ittr = list.iterator(); ittr.hasNext();) {
      int i = ((Integer) ittr.next()).intValue();
      sum += i;
  }
  return sum;
}
	public static int sum(List<Integer> list) {
	  int sum = 0;
	  for (Integer i : list) {
	
	      sum += i;
	  }
	  return sum;
	}