Collections
==========
int x=10; int y=20; int z=30;
In this approach, if i want to keep 10000 values then we can't remember variables to access them.
To resolve this problem we use arrays.

Arrays
=====
It refers to indexed collection of homogenous data elements.
Advantage of Arrays
1. we can represent multiple values by using single varaible, so that readability of the code will be improved.

eg::
```
int arr[] =new int[1000];
```
we resolved the problem, but array is having limitation.

```
Student[] s=new Student[100];
s[0] =new Student();
s[1] =new Employee();//incompatible type: found Employee required:Student
```
==========
To resolve this problem we can use
```
Object[] obj =new Object[1000];
obj[0]=new Student();
obj[1]=new Employee();
```

Limitations of Arrays
==========
1. Array is fixed in size, we can't increase or decrease the size of array.
2. To use the array compulsorily we should know the array size at the begining itself.
3. Array can hold only homogeneous datatype elements.
4. Array is not implemented using standard datastructure,so we don't have ready made methods to perform our task.
   eg: based on some condition, if we want to sort the student object in student[] direct methods are not available so it increases complexity of programmer.

To Overcome the limitations of Arrays we use "Collections".

Collections
==========
1. They are growable in nature(we can increase and decrease)
2. They can hold both hetereogenous and homogenous dataelements
3. Every collection class is implemented using some standard datastructure, so ready methods are available, as a programmer we need to implement rather we should just know how to call those methods.

Which one is prefered over Arrays and Collections?
Arrays is prefered, because performance is good.
Collections is not prefered because
1. List l=new ArrayList(); // default: 10 locations
   if 11th element has to added, then
   a. create a list with 11 locations
   b. copy all the elements from the previous collection
   c. copy the new reference into reference variable
   d. call garbage collector and clean the old memory.

Note: To get something we need to compromise something, so if we use Collections performance is not upto the mark.
Array is language level concept(memory wise it is not good, perforamnce is high)
Collection is API level(memory wise it is good,perfomance is low)

Difference b/w Arrays and Collection
==========
Arrays => It is used only when Array size is fixed
Collection => It is used only when size is not fixed(dynamic)

Arrays => memory wise not recomended to use.
Collection => memory wise recomended to use.

Arrays => Performance wise recomended to use.
Collection => Perforamance wise it is recomended to use.

Arrays => It can hold only homogenous objects
Collection => It can hold both heterogenous and homogenous Objects

Arrays => We can hold both primitive values and Objects
eg: int[] arr=new int[5];
Integer[] arr=new Integer[5];
Collection => It is capable of holding only objects not primitive types.

Arrays => It is not implements using any standard datastructure, so no ready made methods for our requirement,it increases the complexity of programming.

Collection => It is implemented using standard datastructure, so ready made methods are available for our requirement,it is not complex.

What is Collection?
In Order to represent a group of individual object as a single entity then we need to go for Collection.

CollectionFramework
Group of classes and interface, which can be used to represent group of individual object as a single entity, then we need to go for "CollectionFramework".

Java C++
Collection => container
CollectionFramework => STL(standard template library)

To know more information about the framework,then we need to know the specification(interface)
9 key interfaces of Collection framework
a. Collection(I)
b. List(I)
c. Set(I)
d. SortedSet(I)
e. NavigableSet(I)
f. Queue(I)
g. Map(I)
h. SortedMap(I)
i. NavigableMap(I)

Collection

1. In order to represent a group of individual object, then we need to go for "Collection".
2. It is a root interface of collection framework
3. All the commonly used method required for all the collection is a part of Collection(I).

Note: There is no concrete class which would implement the interface Collection(I) directly.

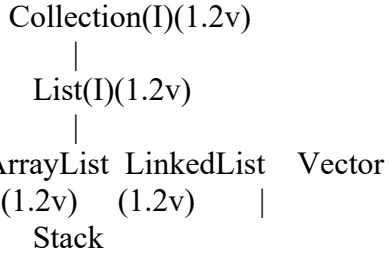
Difference b/w Collection(I) and Collections(C)?

Collection => It is an interface which should be used when we want to represent a group of individual object then we need to go for collection.

Collections => It is a utility class which defines in java.util which defines utility methods for Collection Objects.

List(I)

1. Insertion order must be preserved.
2. Duplicates are allowed.
3. It is the child interface of Collection.

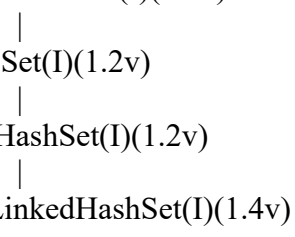


Vector and Stack are a part of jdk1.0 version so they are called as "legacy classes".

Set(I)

It is used to represent a group of individual objects as a single entity such that

1. Duplicates are not allowed
2. Insertion is not preserved
3. It is the child interface of "Collection". then we need to go for "Set".



SortedSet(I)

It is used to represent a group of individual objects as a single entity such that

1. Duplicates are not allowed
2. elements should be added based on some sorting order
3. It is the child interface of "Set". then we need to go for "SortedSet".

Collection(I)(1.2v)

|
Set(I)(1.2v)
|
SortedSet(I)(1.2v)

NavigableSet(I)
1. It is a child class of SortedSet.
2. Various methods are a part of NavigableSet for navigation purpose.
3. Implementation class for NavigableSet is TreeSet.

Collection(I)(1.2v)
|
Set(I)(1.2v)
|
SortedSet(I)(1.2v)
|
NavigableSet(I)(1.6v)
|
TreeSet(C) (1.2V)

What is the difference b/w List and Set?
List => Duplication are allowed, insertion order is preserved.
Set => Duplication are not allowed, insertion order not preserved.

Queue (I):
=> It is the Child Interface of Collection.
=> If we want to Represent a Group of Individual Objects Prior to Processing then we should go for Queue.
Eg: Before sending a Mail we have to Store All MailID's in Some Data Structure and in which Order we added MailID's in the Same Order Only Mails should be delivered (FIFO). For this Requirement Queue is Best Suitable.

Queue
|=> priority queue
|=> Blocking queue
|=> Priority Blocking Queue
|=> Linked Blocking Queue

Note:
=> All the Above Interfaces (Collection, List , Set, SortedSet, NavigableSet,and Queue) Meant for representing a Group of Individual Objects.
=> If we want to Represent a Group of Key - Value Pairs then we should go for Map

Map(I)
a. To represent a group of individual objects as keyvalue pair then we need to opt for Map(I).
Map
|=> HashMap
|=> WeakHashMap
|=> IdentityHashMap
|=> Dictionary
|
Hashtable
|

Properties

8) SortedMap (I):
=> It is the Child Interface of Map.
=> If we want to Represent a Group of Objects as Key- Value Pairs according to Some Sorting Order of Keys then we should go for SortedMap.
=> Sorting should be Based on Key but Not Based on Value

9) NavigableMap (I):
=> It is the Child Interface of SortedMap.
=> It Defines Several Methods for Navigation Purposes.

Legacy Characters
1. Enumeration
2. Dictionary
3. Vector
4. Stack
5. Hashtable
6. Properties

Utility classes
1. Collections
2. Arrays

Cursors
1. Enumeration(I)
2. Iterator(I)
3. ListIterator(I)

Sorting
1.Comparable
2.Comparator

Collection(I)
==========
1. Inside this interface, the commonly used method required for all the collection classes is present

a. boolean add(object o)=> Only one object
b. boolean addAll(Collection c)=>To add group of Object
c. boolean remove(Object o) => to remove particular object
d. boolean removeAll(Collection c)=> to remove particular group of collection
e. void clear() => to remove all the object
f. int size() => to check the size of the array
g. boolean retainAll(Collection c) => except this group of objects remaining all objects should be removed.
h. boolean contains(Object o) => to check whether a particular object exists or not
i. boolean containsAll(Collection c) => To check whether a particular Collection exists or not
j. boolean isEmpty() => To check whether the Collection is empty or not
k. Object[] toArray()=> Convert the object into Array.
l. Iterator iterator() => cursor need to iterate the collection object

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

List(I)
1. It is the child interface of Collection
2. To represent the group of collection objects where
   a. duplicates are allowed(meaning it is stored in index)
   b. insertion order is preserved.(meaning it is stored via index)
3. In list index plays a very important role.

Method assoicated with List(I)
==========
a. void add(int index,Object obj)
b. void addAll(int index,Collection c)
c. Object remove(int index)
d. Object get(int index)
e. Object set(int index,Object o)
f. int indexOf(Object obj)
g. int lastIndexOf(Object obj)
i. ListIterator listIterator()

ArrayList(C)
==========
1. DataStructure: GrowableArray /Sizeable Array
2. Duplicates are allowed through index
3. insertion order is preserved through index
4. Heterogenous objects are allowed.
5. null insertion is also possible.

Constructors
==========
a. ArrayList al=new ArrayList()
Creates an empty ArrayList with the capacity to 10.
   a. if the capacity is filled with 10, then what is the new capacity?
      newcapacity= (currentcapacity * 3/2)+1
      so new capacity is =16,25,38,.....
   b. if we create an ArrayList in the above mentioned order then it would result in performance issue.
   c. To resolve this problem create an ArrayList using 2nd way apporach.

b. ArrayList al=new ArrayList(int initalCapacity)
c. ArrayList l=new ArrayList(Collection c)
   It is used to create an equivalent ArrayList Object based on the Collection Object

eg#1.
```
ArrayList l = new ArrayList();
   l.add("A");
   l.add(10);
   l.add("A");
   l.add(null);
   System.out.println(l); //[A, 10, A, null]

   l.remove(2);
   System.out.println(l); //[A, 10, null]

   l.add(2,"M");
```

```
l.add("N");
System.out.println(l); //[A, 10, M, null, N]
```

Note: Whenever we print any reference it internally calls toString() method.
toString() of all Collection classes is implemented in such a way that it prints the Object in the following order.
o/p => [,,,]
toString() of all Map Object is implemented in such a way that it prints the Object in the following order.
o/p => {k1=v1,k2=v2,k3=v3,....}

Usally we use Collection to store mulitple objects into single entity.
Collection => contanier
To transport the collection over the network,compulsorily the Object should be "Serializable".
1. Every Collection class by default implements Serializable.
2. Every Collection class by default implements Cloneable

ArrayList vs Vector
==========
These 2 classes along with Serializable,Cloneable,it also implements RandomAccess
Any random elements present in ArrayList and Vector can be accesed through same speed, becoz it is accessed using "RandomAccess".
ArrayList and Vector is best suited when our frequent operation is read.

RandomAccess is a marker interface which is a part of java.util package,where the required ability is provided automatically by the jvm.

eg#1.
```
ArrayList l1= new ArrayList();
LinkedList l2=new LinkedList();
System.out.println(l1 instanceof Serializable);//true
System.out.println(l1 instanceof Cloneable);//true
System.out.println(l2 instanceof Serializable);//true
System.out.println(l2 instanceof Cloneable);//true
System.out.println(l1 instanceof RandomAccess);//true
System.out.println(l2 instanceof RandomAccess);//false
```

When to use ArrayList and when not to use?
ArrayList => it is best suited if our frequent operation is "retrieval operation",because it implements RandomAccess interface.

ArrayList => it is the worst choice if our frequent operation is "insert/deletion" in the middle because it should perform so many shift operations.To resolve this problem we should use "LinkedList".

Differences b/w ArrayList and Vector?
ArrayList => Most of the methods are not synchronized.
Vector => Most of the methods are synchronized.

ArrayList =>It is not thread safe becoz mulitple threads can operate on a object.
Vector =>It is thread safe becoz only one thread is allowed to operate.

ArrayList => performance is high becoz threads are not allowed to wait.

Vector => performance is relatively low becoz thread are required to wait.

ArrayList => It is not a legacy class.
Vector => It is a legacy class.

How to use ArrayList, but thread safety is required how would u get or how to get synchronized version of ArrayList ?

```
ArrayList l=new ArrayList();// now 'l' is nonsynchronized
ArrayList l1=Collections.synchronizedList(l);// now 'l1' is synchronized
```

Note::These methods are a part of Collections class(utility class)

```
public static List synchronizedList(List l);
public static Map synchronizedMap(Map m);
public static Set synchronizedSet(Set s);
```

LinkedList
==========
=> Memory management is done effectively if we work with LinkedList.
=> memory is not given in continous fashion.

a. DataStructure is :: doubly linked list
b. heterogenous objects are allowed
c. null insertion is possible
d. duplicates are allowed
e. linkedlist implements Serializable and Cloneable interface but not RandomAccess.

Usage
1. If our frequent operation is insertion/deletion in the middle then we need to opt for "LinkedList".

```
LinkedList l=new LinkedList();
  l.add(a);
  l.add(10);
  l.add(z);
 l.add(2,'a');
 l.remove(3);
```

2. LinkedList is the worst choice if our frequent operation is retreival operation

Constructors
============
a. LinkedList l=new LinkedList();
It creates a empty LinkedList object.
b. LinkedList l=new LinkedList(Collection c);
To convert any Collection object to LinkedList.

Vector
======
Vector:
=> The Underlying Data Structure is Resizable Array ORGrowable Array.
=> Insertion Order is Preserved.
=> Duplicate Objects are allowed.
=> Heterogeneous Objects are allowed.
=> null Insertion is Possible.
=> Implements Serializable, Cloneable and RandomAccess interfaces.
=> Every Method Present Inside Vector is Synchronized and Hence Vector Object is

Thread Safe.
=> Vector is the Best Choice if Our Frequent Operation is Retrieval.
=> Worst Choice if Our Frequent Operation is Insertion OR Deletion in the Middle.

Constructors
==========
1) Vector v = new Vector();
=> Creates an Empty Vector Object with Default Initial Capacity 10.
Once Vector Reaches its Max Capacity then a New Vector Object will be Created with
New Capacity = Current Capacity * 2

2) Vector v = new Vector(intinitialCapacity);
3) Vector v = new Vector(intinitialCapacity, intincrementalCapacity);
4) Vector v = new Vector(Collection c);

Methods:
1) To Add Elements:
add(Object o)-> Collection
add(int index, Object o)-> List
addElement(Object o) -> Vector
2) To Remove Elements:
remove(Object o) -> Collection
removeElement(Object o)-> Vector
remove(int index) -> List
removeElementAt(int index)-> Vector
clear() -> Collection
removeAllElements()-> Vector
3) To Retrive Elements:
Object get(int index)-> List
Object elementAt(int index)-> Vector
Object firstElement() -> Vector
Object lastElement() -> Vector
4) Some Other Methods:
int size()
int capacity()
Enumeration element()

```
import java.util.Vector;
public class VectorDemo {
 public static void main(String[] args) {
  Vector v = new Vector();
  System.out.println(v.capacity()); //10
  for(int i = 1; i<=10; i++) {
   v.addElement(i);
  }
  System.out.println(v.capacity()); //10
  v.addElement("A");
  System.out.println(v.capacity()); //20
  System.out.println(v); //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
 }
}
```

Stack:
=> It is the Child Class of Vector.
=> It is a Specially Designed Class for Last In First Out (LIFO) Order.

Constructor:
Stack s = new Stack();
Methods:
1) Object push(Object o);
To Insert an Object into the Stack.
2) Object pop();
To Remove and Return Top of the Stack.
3) Object peek();
Return Top of the Stack without Removal.
4) boolean empty();
Returns true if Stack is Empty
5) int search(Object o);
Returns Offset if the Element is Available Otherwise Returns -1

```
import java.util.Stack;
public 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
   }
}
```

The 3 Cursors of Java:
=> If we want to get Objects One by One from the Collection then we should go for Cursors.
=> There are 3 Types of Cursors Available in Java.
1) Enumeration
2) Iterator
3) ListIterator

1) Enumeration:
We can Use Enumeration to get Objects One by One from the Collection.
We can Create Enumeration Object by using elements().
public Enumeration elements();
Eg:Enumeration e = v.elements(); //v is Vector Object.
Methods:
1) public boolean hasMoreElements();
2) public Object nextElement();

```
import java.util.*;
public class EnumerationDemo {
 public static void main(String[] args) {
  Vector v = new Vector();
  for(int i=0; i<=10; i++) {
   v.addElement(i);
  }
  System.out.println(v);//[0,1,2,3,4,5,6,7,8,9,10]
  Enumeration e = v.elements();
  while(e.hasMoreElements()) {
   Integer I = (Integer)e.nextElement();
   if(I%2 == 0)
```

```
    System.out.println(I);//0 2 4 6 8 10
   }
   System.out.println(v);[0,1,2,3,4,5,6,7,8,9,10]
  }
}
```

Limitations of Enumeration:

=> Enumeration Concept is Applicable Only for Legacy Classes and it is Not a Universal Cursor.

=> By using Enumeration we can Perform Read Operation and we can't Perform Remove Operation.

To Overcome Above Limitations we should go for Iterator.

2) Iterator:

=> We can Use Iterator to get Objects One by One from Collection.

=> We can Apply Iterator Concept for any Collection Object. Hence it is Universal Cursor.

=> By using Iterator we can Able to Perform Both Read and Remove Operations.

=> We can Create Iterator Object by using iterator() of Collection Interface.

```
 public Iterator iterator();
```

Eg:Iterator itr = c.iterator(); //c Means any Collection Object.

Methods:

1) public booleanhasNext()
2) public Object next()
3) public void remove()

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

Limitations:

=> By using Enumeration and Iterator we can Move Only towards Forward Direction and we can’t Move Backward Direction. That is these are Single Direction Cursors but NotBi Direction.

=> By using Iterator we can Perform Only Read and Remove Operations and we can't Perform Addition of New Objects and Replacing Existing Objects.

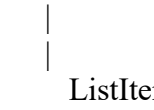
To Overcome these Limitations we should go for ListIterator.

3) ListIterator:

=> ListIterator is the Child Interface of Iterator.
=> By using ListIterator we can Move Either to the Forward Direction OR to the Backward Direction. That is it is a Bi-Directional Cursor.
=> By using ListIterator we can Able to Perform Addition of New Objects andReplacing existing Objects. In Addition to Read and Remove Operations.
=> We can Create ListIterator Object by using listIterator().
public ListIteratorlistIterator();
Eg:ListIteratorltr = l.listIterator(); //l is Any List Object

Methods:

=> ListIteratoris the Child Interface of Iterator and Hence All Iterator Methods by Default Available to the ListIterator.



ListIteratorDefines the following 9 Methods.

```
public boolean hasNext()
public Object next()
public int nextIndex()

public boolean hasPrevious()
public Object previous()
public int previousIndex()

public void remove()
public void set(Object new)
public void add(Object new)
```

```
import java.util.*;
public class ListIteratorDemo {
 public static void main(String[] args) {
  LinkedList l = new LinkedList();
  l.add("sachin");
  l.add("kohli");
  l.add("yuvi");
  l.add("dhoni");
  System.out.println(l);//[sachin,kohli,yuvi,dhoni]
 ListIteratorltr = l.listIterator();
 while(ltr.hasNext()) {
  String s = (String)ltr.next();
  if(s.equals("dhoni"))
   ltr.remove();
  if(s.equals("sachin"))
   ltr.add("ponting");
  if(s.equals("yuvi"))
   ltr.add("gilchrist");
 }
 System.out.println(l);//[sachin,ponting,kohli,yuvi,gilchrist]
   }
}
```

Note:

The Most Powerful Cursor is ListIterator. But its Limitation is, it is Applicable Only for List Objects.

refer:cursorschart.png

Internal Implementation of Cursors
==========
```
import java.util.*;
public class CursorDemo {
 public static void main(String args[]) {
  Vector v = new Vector();
  Enumeration e = v.elements();
  Iterator itr  = v.iterator();
  ListIterator litr = v.listIterator();
  System.out.println(e.getClass().getName());//java.util.Vector$1
  System.out.println(itr.getClass().getName());//java.util.Vector$itr
  System.out.println(litr.getClass().getName());//java.util.Vector$Listitr
      }
}
```

Set:
=> It is the Child Interface of Collection.
=> If we want to Represent a Group of Individual Objects as a Single Entity where Duplicates are Not allowed and Insertion Order is Not Preserved then we should go for Set.
=> Set Interface doesn't contain any New Methods and Hence we have to Use Only Collection Interface Methods
refer:setchart.png

HashSet
=======
1. Duplicates are not allowed,if we try to add it would not throw any error rather it would return false.
2. Internal DataStructure: Hashtable
3. null insertion is possible.
4. heterogenous data elements can be added.
5. If our frequent operation is search, then the best choice is HashSet.
6. It implements Serializable,Cloneable, but not random access.

Constructors
==========
HashSet s=new HashSet(); Default initial capacity is 16
Default FillRation/load factor is 0.75

Note: In case of ArrayList, default capacity is 10, after filling the complete capacity then new ArrayList would be created.

In case of HashSet, after filling 75% of the ratio only new HashSet will be created.

HashSet s=new HashSet(int intialiCapacity);//specified capacity with default fill ration=0.75
HashSet s=new HashSet(int initaliCapacity,float fillRatio)
HashSet s=new HashSet(Collection c);

LoadFactor
=========
After loading how much ratio,a new object will be created is called as "LoadFactor".

Program

=======

```
import java.util.HashSet;
public class Test  {
 public static void main(String[] args){
  HashSet h=new HashSet();
  h.add("D");
  h.add("B");
  h.add("C");
  h.add("Z");
  h.add(null);
  h.add(10);
  System.out.println(h);//[Z,null,C,D,B,10]
  System.out.println(h.add("Z"));//false
     }
}
```

LinkedHashSet

==========

It is the child class of "HashSet".
DataStructure: HashTable + linkedlist
duplicates   : not allowed
insertion order: preserved
null allowed   : yes

All the constructors and methods which are a part of HashSet will be a part of "LinkedHashSet",but except "insertion order will be preserved".

Difference b/w HashSet and LinkedHashSet

HashSet   => underlying datastructure is "HasTable"
LinkedHashSet => underlying datastructure is combination of "Hashtable + "linkedlist" .

HashSet   => Duplicates are not allowed and insertion order is not preserved
LinkedHashSet => Duplicates are not allowed,but insertion order is preserved.

HashSet   => 1.2V
LinkedHashSet => 1.4v

```
import java.util.LinkedHashSet;

public class Test  {
 public static void main(String[] args){
  LinkedHashSet h=new LinkedHashSet();
  h.add("D");
  h.add("B");
  h.add("C");
  h.add("Z");
  h.add(null);
  h.add(10);
  System.out.println(h);//[D,B,C,Z,null,10]

  System.out.println(h.add("Z"));//false
     }
}
```

Note: insertion order is preserved, but duplicates are not allowed.
Whenever we want to develp cache based application,where duplicates are not allowed insertion order must be preserved then we go for "LinkedHashSet".

SortedSet(I)
==========
It is the child interface of Set
Group of individual objects, where duplicates are not allowed,but the elements should be sorted in some order.
eg: {3,2,1}
{1,2,3}
{3,1,2}
{2,1,3}

Some specific methods w.r.t SortedSet are
a. Object firstElement() => returns first element
b. Object lastElement() => returns last element
c. SortedSet headSet(Object obj)=> returns sortedset whose elements are < obj
d. SortedSet tailSet(Object obj)=> returns sortedset whose elements are >=obj
e. SortedSet subSet(Object obj1,Object obj2)=> returns subset whose elements are >=obj1 and <obj2
f. Compartor compartor() => returns Comparotor object that describes underlying sorting technique
default natural sorting order means it returns null.

String object=> default natural sorting order is Alphabetical order
Numbers => default natural sorting order is Ascending order.

SortedSet
100 101 102 103 104 105 106 107
firstElement() => 100
lastEleemnt() => 107
headSet(104) => 100 101 102 103
tailSet(105) => 105,106,107
subset(101,104) => 101,102,103
compartor() => null

TreeSet
=======
Underlying Datastructure: BalancedTree
duplicates : not allowed
insertion order : not preserved
heterogenous element: not possible,if we try to do it would result in "ClassCastException".
null insertion : possible only once
Implements Serializable and Cloneable interface,but not RandomAccess.
All Objects will be inserted based on "some sorting order" or "customized sorting order".

Constructor
==========
TreeSet t=new TreeSet();//All objects will be inserted based on some default natural sorting order.

TreeSet t=new TreeSet(Comparator); //All objects will be inserted based on some customized sorting order.

TreeSet t=new TreeSet(Collection c);
TreeSet t=new TreeSet(SortedSet)

Note::
Comparable => Default natural sorting order
Comparator => Customized sorting order.

Program
=======
```
import java.util.TreeSet;
public class Test  {
 public static void main(String[] args){
  TreeSet t=new TreeSet();
  t.add("A");
  t.add("a");
  t.add("B");
  t.add("Z");
  t.add("L");

  System.out.println(t);//ABZLa

  t.add(10);//ClassCastException
     }
}
```

nullacceptance
==========
1. For non-empty treeset if we try to add null, then it would result in "NullPointerException".
2. For empty tree set the 1st Element null insertion is possible,but after inserting null if we try to insert any another element then
   it would result in "NullPointerException".

Till JDK1.6 as the first element, we can add null but from jdk1.7onwards null insertion as the first element is also no t allowed.

Program
=======
```
import java.util.TreeSet;
public class Test  {
 public static void main(String[] args){
  TreeSet t=new TreeSet();
  t.add("a");
  t.add("b");
        t.add("c");
  System.out.println(t);
  t.add(null);//NullPointerException
     }
}
```

eg#2.
```
importjava.util.TreeSet;
classTreeSetDemo {
public static void main(String[] args) {
  TreeSet t = new TreeSet();
  t.add(new StringBuffer("A"));
  t.add(new StringBuffer("Z"));
  t.add(new StringBuffer("L"));
```

```
  t.add(new StringBuffer("B"));
  System.out.println(t);
 }
}
```

Output:ClassCastException
Reason:: TreeSet t=new TreeSet()
a. we inform jvm to use default natural sorting order
To sort the elements must be
a. Homogenous
b. Comparable(class should implement Comparable)
otherwise it would result in "ClassCastException".

Note:: Object is said to be Comparable, iff the corresponding class implements "Comparable".
All Wrapper class and String class implements Comparable so we can compare the objects.

Comparable(I)
=> It is a part of java.lang package
=> It contains only one method compareTo.
public int compareTo(Object o)

=> obj1.compareTo(obj2)
returns -ve iff obj1 has to come before obj2
returns +ve iff obj1 has to come after obj2
returns 0 if both are equal

eg#1.

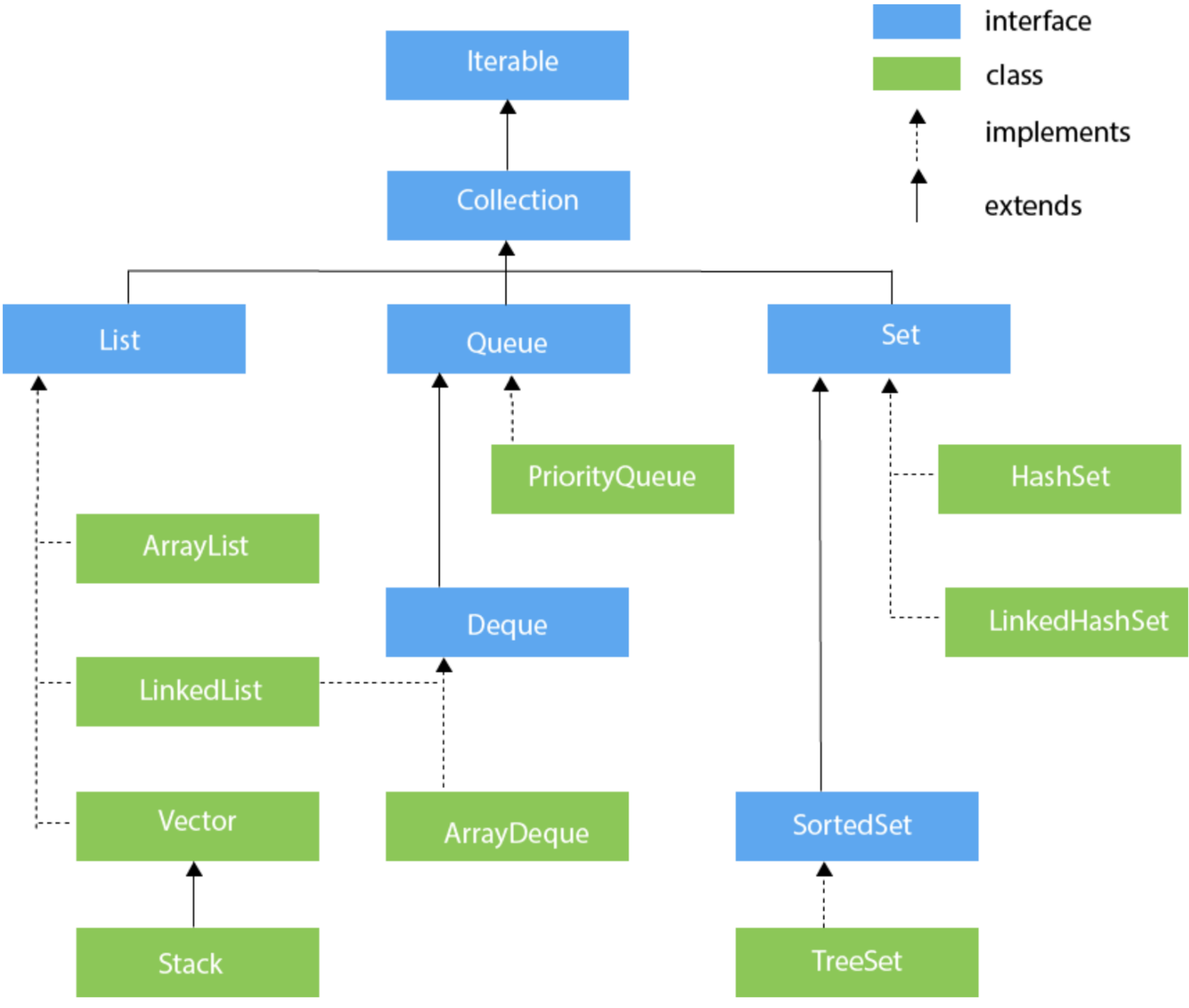
```
System.out.println("A".compareTo("Z"));//A should  come before Z so -ve
System.out.println("Z".compareTo("K"));//Z should  come after  K so +ve
System.out.println("A".compareTo("A"));//Both are equal zero
System.out.println("A".compareTo(null));//NullPointerException
```

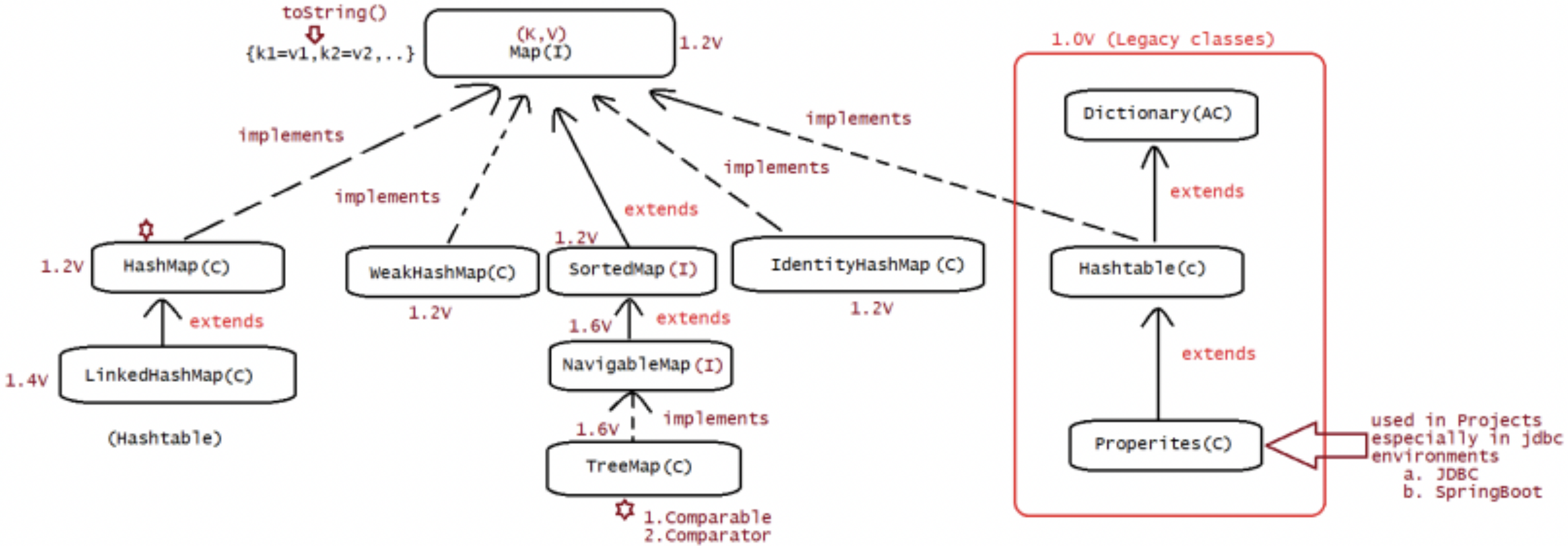
eg#2.

```
import java.util.TreeSet;
public class TestApp{
 public static void main(String... args){
  TreeSet ts= new TreeSet();

  ts.add("K");
  ts.add("Z");//internally  "Z".compareTo("K") +ve
  ts.add("A");//internally  "A".compareTo("K") -ve
  ts.add("A");//internally  "A".compareTo("K") -ve
            //internally  "A".compareTo("A")  0

  System.out.println(ts);//[A K Z]
 }
}
```

Rule: obj1.compareTo(obj2)
obj1 => The object which needs to be inserted.
obj2 => The object which is already inserted.

Whenever we are depending on default natural sorting order,if we try to insert the elements
then internally it calls compareTo() to IdentifySorting order.


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet


toString()
{k1=v1,k2=v2,..}
(K,V)
Map(I)
1.2V
implements
implements
implements
implements
1.2V
HashMap(C)
extends
1.4V
LinkedHashMap(C)
(Hashtable)
WeakHashMap(C)
1.2V
1.2V
SortedMap(I)
extends
1.6V
NavigableMap(I)
implements
1.6V
TreeMap(C)
1.Comparable
2.Comparator
IdentityHashMap(C)
1.2V
1.0V (Legacy classes)
Dictionary(AC)
extends
Hashtable(c)
extends
Properites(C)
used in Projects
especially in jdbc
environments
a. JDBC
b. SpringBoot

class	Internal DS	preserve insertion order	permits null value	Duplicate
ArrayList	Dynamic Array DS	✓	✓	✓
LinkedList	Doubly Linked list	✓	✓	✓
ArrayDeque	doubleendedQueue	✓	✗	✓
PriorityQueue	min-heap	✗	✗	✓
TreeSet	Binary search Tree	✗	✗	✗
HashSet	HashTable (Hashing)	✗	✓	✗
LinkedHashSet	HashTable	✓	✓	✗

	for( ) loop	for-each	Iterator	List iterator	descending iterator	Enumeration
ArrayList	✓	✓	✓	✓	✗	✗
LinkedList	✓	✓	✓	✓	✓	✗
ArrayDeque	✗	✓	✓	✗	✓	✗
PriorityQueue	✗	✓	✓	✗	✗	✗
TreeSet	✗	✓	✓	✗	✓	✗
Hashset	✗	✓	✓	✗	✗	✗
LinkedHashset	✗	✓	✓	✗	✗	✗

Property	Enumeration	Iterator	ListIterator
Applicable For	Only Legacy Classes	Any Collection Objects	Only List Objects
Movement	Single Direction (Only Forward)	Single Direction (Only Forward)	Bi-Direction
How To Get	By using elements()	By using iterator()	By using listIterator() of List (I)
Accessability	Only Read	Read and Remove	Read , Remove, Replace And Addition of New Objects
Methods	hasMoreElements() nextElement()	hasNext() next() remove()	9 Methods
Is it legacy?	Yes (1.0 Version)	No (1.2 Version)	No (1.2 Version)

Property	Fail Fast	Fail Safe
Does it through ConcurrentModificationException?	Yes	No
Is the Cloned Copy will be Created?	No	yes
Memory Problems	No	yes
Examples	ArrayList, Vector, HashMap, HashSet	ConcurrentHashMap, CopyOnWriteArrayList CopyOnWriteArraySet

Map
===
=> It is not a child interface of Collection.
=> If we want to represent group of Objects as key-value pair then we need to go for Map.
=> Both keys and values are Objects only
=> Duplicate keys are not allowed but values are allowed.
=> Key-value pair is called as "Entry".

refer: MapHierarchy.png

Map methods
---
1. It contains 12 methods which is common for all the implementation Map Objects
a. Object put(Object key,Object value) // To add key,value pair
b. void putAll(Map m) // To add another map
c. Object get(Object key) // To get the value based on key
d. Object remove(Object key) //To remove an entry based on key
e. boolean containsKey(Object key) //Check whether it contains key or not
f. boolean containsValue(Object value)//Check whether it contains value or not
g. boolean isEmpty() //To check wheter the Map is empty or not
h. int size() //To get the size of a Map
i. void clear() //To remove all Entry from a map

views of a Map
j.Set keySet() //Convert the key's of Map into Set for reading purpose
k.Collection values() //Convert the values of Map into Collection for reading purpose
l.Set entrySet() // Convert whole Entry of Map into Set for reading purpose.

```
Entry(I)
=======
 1. Each key-value pair is called Entry.
 2. Without existence of Map,there can't be existence of Entry Object.
 3. Interface entry is defined inside Map interface.

interface Map{
      interface Entry{
            Object getKey();  //To get the key using Map.Entry Object
            Object getValue();//To get the value using Map.Entry Object
            Object setValue(Object newValue);//To update the value Using Map.Entry
Object
      }
}

HashMap
=======
 Underlying DataStructure: Hashtable
 insertion order          : not preserved
 duplicate keys           : not allowed
```

```
 duplicate values         : allowed
 Heterogenous objects     : allowed
 null insertion           : for keys allowed only once,but for values can be any no.
```

Difference b/w HashMap(c) and Hashtable(c)
==========
HashMap => All the methods are not synchronized.
Hashtable => All the methods are synchronzied.

HashMap => At a time multiple threads can operate on a Object, so it is not ThreadSafe.
Hashtable => At a time only one Thread can operate on a Object, so it is ThreadSafe.

HashMap => Pefromance is high.
Hashtable => Performance is low.

HashMap => null is allowed for both keys and values.
Hashtable => null is not allowed for both keys and values,it would result in NullPointerException.

HashMap => Introduced in 1.2v
Hashtable => Introduced in 1.0v

Constructors
==========
1. HashMap hm=new HashMap()
//default capacity => 16, loadfactor => 0.75(upon increase of data by 75% automatically

size of HashMap will be doubled)

2. Hashmap hm=new HashMap(int capacity);

3. HashMap hm=new HashMap(int capacity,float fillration);

4. HashMap hm=new HashMap(Map m);

```
LinkedHashMap
=============
 => It is the child class of HashMap.
 => It is same as HashMap,but with the following difference

       HashMap             => underlying datastructure is hashtable.
       LinkedHashMap => underlying datastructure is LinkedList + hashtable.

       HashMap             => insertion order not preserved.
       LinkedHashMap => insertion order preserved.

       HashMap             => introduced in 1.2v
       LinkedHashMap => introduced in 1.4v
```

WeakHashMap
==========
 It is exactly same as HashMap, with the following differences.
1. HashMap will always dominate Garbage Collector, that is if the Object is a part of HashMap
   and if the Object is Garbage Object, still Garbage Collector won't remove that Object from
   heap since it is a part of HashMap. HashMap dominates GarbageCollector.
2. Garbage Collector will dominate WeakHashMap, that is if the Object is part of WeakHashMap and
   if that Object is Garbage Object, then immediately Garbage Collector will remove that Object
   from heap even though it is a part of WeakHashMap, so we say Garbage Collector dominates
   "WeakHashMap".

Hashtable:
=> The Underlying Data Structure for Hashtable is Hashtable Only.
=> Duplicate Keys are Not Allowed. But Values can be Duplicated.

=> Insertion Order is Not Preserved and it is Based on Hashcode of the Keys.
=> Heterogeneous Objects are Allowed for Both Keys and Values.
=> null Insertion is Not Possible for Both Key and Values. Otherwise we will get Runtime
Exception Saying NullPointerException.
=> It implements Serializable and Cloneable,but not RandomAccess.
=> Every Method Present in Hashtable is Synchronized and Hence Hashtable Object is Thread
Safe.

Constructors:
1) Hashtable h = new Hashtable();
Creates an Empty Hashtable Object with Default Initial Capacity 11 and
Default Fill Ratio 0.75.
2) Hashtable h = new Hashtable(intinitialcapacity);
3) Hashtable h = new Hashtable(intinitialcapacity, float fillRatio);
4) Hashtable h = new Hashtable(Map m);

Note:
If we are Depending on Default Natural Sorting Order Compulsory Objects should be Homogeneous and Comparable.
Otherwise we will get RE: ClassCastException.
An object is said to be Comparable if and only if corresponding class implements Comparable interface.
All Wrapper Classes, String Class Already Implements Comparable Interface. But StringBuffer Class doesn't Implement Comparable Interface.

```
@FunctionalInterface
public interface Comparable<T> {
  public abstract int compareTo(T);
}
```

```
obj1.compareTo(obj2)
                  |=> returns -ve value, if obj1 has to come before obj2
                  |=> returns +ve value, if obj1 has to come after  obj2
                  |=> returns 0 value, if both obj1 and obj2 are equal
```

```
import java.util.*;
class Test {
      public static void main(String[] args) {
            TreeSet ts =new TreeSet();

            ts.add("A");
            ts.add("Z");
            ts.add("L");
            ts.add("K");
            ts.add("B");

            System.out.println(ts);

      }
}
```

Comparable (I):
Comparable Interface Present in java.lang Package and it contains Only One Method compareTo().

```
      obj1.compareTo(obj2)
            Returns –ve if and Only if obj1 has to Come Before obj2.
            Returns +ve if and Only if obj1 has to Come After obj2.
            Returns 0 if and Only if obj1 and obj2 are Equal.
```

```
eg#1.
System.out.println("A".compareTo("Z")); //-ve value
System.out.println("Z".compareTo("K")); // +value
System.out.println("Z".compareTo("Z")); // zero
System.out.println("Z".compareTo(null));//NPE
```

Wheneverwe are Depending on Default Natural Sorting Order and if we are trying to Insert Elements then Internally JVM will
Call compareTo() to IdentifySorting Order.

```
      TreeSet t = new TreeSet();
            t.add(“K”);
            t.add(“Z”); “Z”.compareTo(“K”);
            t.add(“A”); “A”.compareTo(“K”);
            t.add(“A”); “A”.compareTo(“A”);
            System.out.println(t);//[A,K,Z] => Sorting is ascending order
```

Note:
  For String default natural sorting order is   "Ascending order".
  For Number default natural sorting order is "Ascending order"

Comparator(I)
============
Note: If we are Not satisfied with Default Natural Sorting Order OR if Default Natural Sorting Order is Not Already Available then
           we can Define Our Own Sorting by using Comparator Object.

```
public interface java.util.Comparator<T> {
  public abstract int compare(T, T);
  public abstract boolean equals(java.lang.Object);
}
```

Comparator (I):
This Interface Present in java.util Package.
Methods: It contains 2 Methodscompare() and equals().

```
public  int compare(Object obj1, Object obj2);
      Returns –ve if and Only if obj1 has to Come Before obj2.
      Returns +ve if and Only if obj1 has to Come After obj2.
      Returns 0 if and Only if obj1 and obj2 are Equal.

public boolean equals(Object o);
      Whenever we are implementing Comparator Interface Compulsory we should Provide Implementation for compare().
      Implementing equals() is Optional because it is Already Available to Our Class from Object Class through Inheritance.

import java.util.*;
class     TreeSetDemo {
public static void main(String[] args) {
            TreeSet t = new TreeSet(new MyComparator());//line-1
                  t.add(10);
                  t.add(0);
                  t.add(15);
                  t.add(5);
                  t.add(20);
                  t.add(20);
                  System.out.println(t);//[20, 15, 10, 5, 0]
      }
}
class  MyComparator implements Comparator {
            public  int compare(Object obj1, Object obj2) {
                        Integer i1 = (Integer)obj1;
                        Integer i2 = (Integer)obj2;
                        if(i1 < i2)
                              return +1;
                        else if(i1 > i2)
                              return -1;
                        else
                              return 0;
            }
}
```

At Line 1 if we are Not Passing Comparator Object as an Argument then Internally JVM will Call compareTo(),
 Which is Meant for Default Natural Sorting Order (Ascending Order).

In this Case the Output is [0, 5, 10, 15, 20].

At Line 1 if we are Passing Comparator Object then JVM will Call compare() Instead of compareTo().
Which is Meant for Customized Sorting (can be Ascending /Descending Order).
In this Case the Ouput is [20, 15, 10, 5, 0]

Various Possible Implementations of compare():
==========

```
public int compare(Object obj1, Object obj2) {
            Integer I1 = (Integer)obj1;
            Integer I2 = (Integer)obj2;
            return I1.compareTo(I2);
            return -I1.compareTo(I2);
            return I2.compareTo(I1);
            return -I2.compareTo(I1);
            return +1;
            return -1;
            return 0;
}
```

Output:

1. Ascending order
2. Descending order
3. Descending order
4. Ascending order
5. insertion order
6. reverse of insertion order
7. only first element will be inserted.

Write a Program to Insert String Objects into the TreeSet where the Sorting Order is of Reverse of Alphabetical Order:

```
import java.util.*;
class  TreeSetDemo {
      public static void main(String[] args) {
                  TreeSet t = new TreeSet(new MyComparator());
                  t.add("sachin");
                  t.add("ponting");
                  t.add("sangakara");
                  t.add("fleming");
                  t.add("lara");
                  System.out.println(t);
      }
}
class   MyComparator implements Comparator {
            public  int compare(Object obj1, Object obj2) {
                        String s1 = obj1.toString();
                        String s2 = (String)obj2;
                  return s2.compareTo(s1);
                  //return -s1.compareTo(s2);
      }
}
```

Write a Program to Insert StringBuffer Objects into the TreeSet where Sorting Order is Alphabetical Order:

```
import  java.util.*;
class  TreeSetDemo {
            public static void main(String[] args) {
```

```
                        TreeSet t = new TreeSet(new MyComparator1());
                        t.add(new StringBuffer("A"));
                        t.add(new StringBuffer("Z"));
                        t.add(new StringBuffer("K"));
                        t.add(new StringBuffer("L"));
                        System.out.println(t);
      }
}
class MyComparator1 implements Comparator {
            publicint compare(Object obj1, Object obj2) {
                        String s1 = obj1.toString();
                        String s2 = obj2.toString();
                        return s1.compareTo(s2); //[A, K, L, Z]
      }
}
```

Write a Program to Insert String and StringBuffer Objects into the TreeSet where Sorting Order is Increasing Length Order.
If 2 Objects having Same Length then Consider their Alphabetical Order:
eg: A,ABC,AA,XX,ABCE,A

output: A,AA,XX,ABC,ABCE

```
import  java.util.*;
class  TreeSetDemo {
public static void main(String[] args) {
                  TreeSet t = new TreeSet(new MyComparator());
                  t.add("A");
                  t.add(new StringBuffer("ABC"));
                  t.add(new StringBuffer("AA"));
                  t.add("XX");
                  t.add("ABCE");
                  t.add("A");
                  System.out.println(t);
      }
}
class  MyComparator implements Comparator {
      public      int compare(Object obj1, Object obj2) {
            String s1 = obj1.toString();
            String s2 = obj2.toString();
            int i1 = s1.length();
            int i2 = s2.length();
            if(i1 < i2)return -1;
            else if(i1 > i2)return 1;
            else  return s1.compareTo(s2);
      }
}
```

Note:
 if we are use TreeSet(), then the condition is
      a. Object should be homogenous.
      b. Object should be comparable(class should implement Comparable(I)).

if we are use TreeSet(Comparator c) then what is the condition?
      a. Object need not be homogenous.
      b. Object need not implement Comparable.

When to go for Comparable interface and When to go Comparator interface?
Ans. Predefined Comparable classes like String,Wrapper class ====> Default natural sorting is already available
if we are not happy with natural sorting order, we want customization then we need to go for "Comparator(I)".

For Predefined Non-Comparable class like StringBuffer => Comparator(I) is used for both natural sorting order and

and customized sorting order.

For userdefined class like Employee,Student =====> Developer if he comes up with own logic of sorting,then he should

implement Comparable(I) and give it as a ready made logic.

Nitin.M
======

```
class Employee implements Comparable
{
	int id;
	String name;
	int age;

	public int compareTo(Object obj){
		//sorting is done based on "id"
		;;;;;
	}
}
```

If the developer who is using Employee class, if he is not interested with sorting based on "id" given by the api, then he can
use "Comparator".

When we go for Comparable and When we go for Comparator:
Comparable Vs Comparator:

=> For Predefined Comparable Classes (Like String) Default Natural Sorting Order is Already Available. If we are Not satisfied
with that we can Define Our Own Sorting by Comparator Object.
=> For Predefine Non- Comparable Classes (Like StringBuffer) Default Natural Sorting Order is Not Already Available.
If we want to Define Our Own Sorting we can Use Comparator Object.
=> For Our Own Classes (Like Employee) the Person who is writing Employee Class he is Responsible to Define Default Natural
Sorting Order by implementing Comparable Interface.
=> The Person who is using Our Own Class if he is Not satisfied with Default Natural Sorting Order he can Define his Own
Sorting by using Comparator Object.
If he is satisfied with Default Natural Sorting Order then he can Use Directly Our Class.

Write a Program to Insert Employee Objects into the TreeSet where DNSO is Based on Ascending Order of EmployeeId and
Customized Sorting Order is Based on Alphabetical Order of Names:

DNSO -> Default Natural Sorting Order

```
import  java.util.*;
class Employee implements Comparable {
```

```
            String name;
            int   eid;
            Employee(String name, inteid) {
                  this.name = name;
                  this.eid = eid;
            }
      public String toString() { return name+"-----"+eid;}
      public int compareTo(Object obj) {
            int eid1 = this.eid;
            Employee e = (Employee)obj;
            int eid2 = e.eid;
            if(eid1 < eid2) return -1;
            else if(eid1 > eid2) return 1;
            else return 0;
      }
}
class     Test {
      public static void main(String[] args) {

            Employee e1 = new Employee("sachin", 10);
            Employee e2 = new Employee("ponting", 14);
            Employee e3 = new Employee("lara", 9);
            Employee e4 = new Employee("flintoff", 17);
            Employee e5 = new Employee("anwar", 23);

            TreeSet t = new TreeSet();
            t.add(e1);
            t.add(e2);
            t.add(e3);
            t.add(e4);
            t.add(e5);
            System.out.println(t);

            TreeSet t1 = new TreeSet(new MyComparator());
            t1.add(e1);
            t1.add(e2);
            t1.add(e3);
            t1.add(e4);
            t1.add(e5);
            System.out.println(t1);
      }
}
class    MyComparator implements Comparator {
      public      int compare(Object obj1, Object obj2) {
            Employee e1 = (Employee) obj1;
            Employee e2 = (Employee) obj2;
            String s1 = e1.name;
            String s2 = e2.name;
            return s1.compareTo(s2);
      }
}
```

Comparison of Comparable and Comparator:

Comparable(I)
Present in java.lang Package
It is Meant for Default Natural Sorting Order.
Defines Only One Method compareTo()

All Wrapper Classes and String Class implements Comparable Interface.

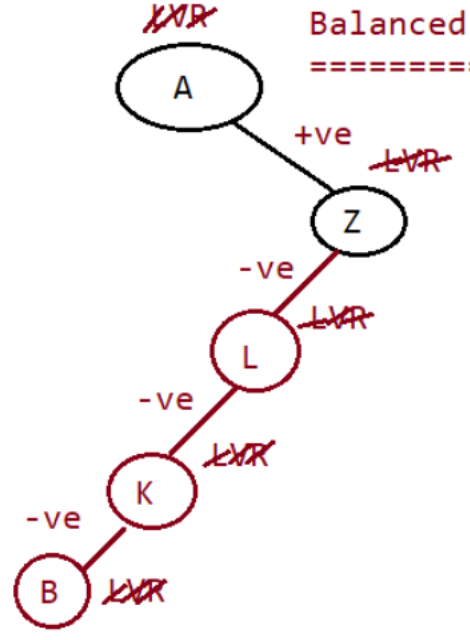
Comparator(I)
Present in java.util Package
It is Meant for Customized Sorting Order.
Defines 2 Methods compare() and equals().
The Only implemented Classes of Comparator are Collator and RuleBaseCollator.

```
System.out.println("A".compareTo("Z"));// -ve value
System.out.println("Z".compareTo("K"));// +ve value
System.out.println("Z".compareTo("Z"));// 0
```

Rules while constructing a binary tree

-ve means in binary tree the node should be to the left
+ve means in binary tree the node should be to the right
zero means in binary tree the nodes are duplicated

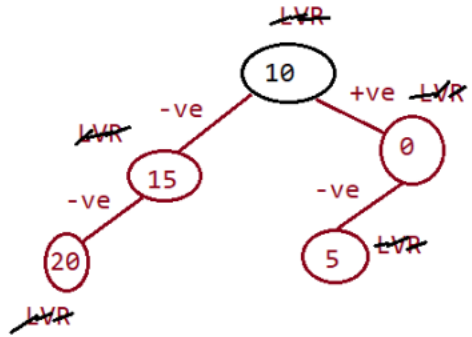
```
TreeSet ts =new TreeSet();
    ts.add("A");
    ts.add("Z"); "Z".compareTo("A");
    ts.add("L"); "L".compareTo("A");"L".compareTo("Z");
    ts.add("K"); "K".compareTo("A");"K".compareTo("Z");
                                    "K".compareTo("L");

   ts.add("B"); "B".compareTo("A"); "B".compareTo(Z)
                "B".compareTo("L"); "B".compareTo("K");

System.out.println(ts); [ A B K L Z ]
```



Balanced Binary Tree
==========

A B K L Z

```
TreeSet ts =new TreeSet(new MyCompartor());
   ts.add(10);
   ts.add(0); compare(0,10)
   ts.add(15); compare(15,10)
   ts.add(5); compare(5,10)
              compare(5,0)
   ts.add(20); compare(20,10)
               compare(20,15)
   ts.add(20); compare(20,10)
               compare(20,15)
               compare(20,20)

         20 15 10 5 0
```




```
compare(Object obj1,Object obj2)
 |=> returns -ve value, if obj1 has to come before obj2
 |=> returns +ve value, if obj1 has to come after  obj2
 |=> returns 0 value, if both obj1 and obj2 are equal
```

```
if (i1<i2)
    return +1;  <= reversing the logic
else if(i1>i2)
     return -1;  <= reversing the logic
else
     return 0;
```

Generic classes:
Until 1.4v a non-generic version of ArrayList class is declared as follows.

Example:
```
class ArrayList{
      add(Object o);
      Object get(int index);
}
```
add() method can take object as the argument and hence we can add any type of object to the ArrayList.
Due to this we are not getting type safety.
The return type of get() method is object hence at the time of retrieval compulsory we should perform type casting.

But in 1.5v a generic version of ArrayList class is declared as follows.

```
                      |=> Type parameter
class ArrayList<T>{
      add(T t);
      T get(int index)
}
```

Based on our requirement T will be replaced with our provided type.
For Example to hold only string type of objects we can create ArrayList object as follows.

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

For this requirement compiler considered ArrayList class is
Example:
```
class ArrayList<String>{
            add(String s);
            String get(int index);
}
```
add() method can take only string type as argument hence we can add only string type of objects to the List.
By mistake if we are trying to add any other type we will get compile time error.

eg#1.
```
ArrayList<String> al =new ArrayList<String>();
      al.add("NavinReddy");
      al.add(10);//CE: can't find symbol
                              symbol:   method add(int)
                              location : class
java.util.ArrayList<java.lang.String>
                                                  al.add(10)
```

eg#2.
```
ArrayList<String> al =new ArrayList<String>();
      al.add("NavinReddy");
String name = al.get(0);//type casting is not requried
```

Hence through generics we are getting type safety.
At the time of retrieval it is not required to perform any type casting we can assign its values directly to string variables.

In Generics we are associating a type-parameter to the class, such type of parameterised classes are nothing but Generic
classes.

Generic class : class with type-parameter.

Based on our requirement we can create our own generic classes also.
Example:

```
class Account<T>
{}
Account<Gold> g1=new Account<Gold>();
Account<Silver> g2=new Account<Silver>();
```

Example:

```
class Gen<T>{

      T obj;
      Gen(T obj){
            this.obj=obj;
      }
      public void show(){
                  System.out.println("The type of object
is :"+obj.getClass().getName());
      }
      public T getObject(){
                  return obj;
      }
}
class GenericsDemo{
      public static void main(String[] args){
                  Gen<Integer> g1=new Gen<Integer>(10);
                  g1.show();
                  System.out.println(g1.getObject());

                  Gen<String> g2=new Gen<String>("iNeuron");
                  g2.show();
                  System.out.println(g2.getObject());

                  Gen<Double> g3=new Gen<Double>(10.5);
                  g3.show();
                  System.out.println(g3.getObject());
      }
}
```

Output:

```
The type of object is: java.lang.Integer
10

The type of object is: java.lang. String
iNeuron

The type of object is: java.lang. Double
10.5
```

Note: To get the underlying object of any reference type we use a method called ref.getClass().getName()

eg1

```
interface Calculator{}
class Casio implements Calculator{}
class Quartz implements Calculator{}
class Kadio implements Calculator{}
```

```
Calculator c1 =new Casio();
System.out.println(c1.getClass().getName());//Casio

Calculator c2 =new Quartz();
System.out.println(c2.getClass().getName());//Quartz

Calculator c3 =new Kadio();
System.out.println(c3.getClass().getName());//Kadio
```

Bounded types

We can bound the type parameter for a particular range by using extends keyword such types are called bounded types.

Example 1:

```
class Test<T>
{}
Test <Integer> t1=new Test< Integer>();
Test <String> t2=new Test < String>();
```

Here as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

Example 2:

```
class Test<T extends X>
{}
```

If x is a class then as the type parameter we can pass either x or its child classes.

If x is an interface then as the type parameter we can pass either x or its implementation classes.

eg#1.

```
class Test <T extends Number>{}
class Demo{
      public static void main(String[] args){
            Test<Integer> t1 = new Test<Integer>();
            Test<String>   t2 = new Test<String>(); //CE
      }
}
```

eg#2.

```
class Test <T extends Runnable>{}
class Demo{
      public static void main(String[] args){
            Test<Thread> t1 = new Test<Thread>();
            Test<String>   t2 = new Test<String>(); //CE
      }
}
```

Keypoints about bounded types

=> We can't define bounded types by using implements and super keyword

=> But implements keyword purpose we can replace with extends keyword.

```
      eg: class Test<T implements Runnable>{}//invalid
            class Test<T super String>{}//invalid
```

```
keyPoints abount BoundedTypes
==========================

=> As the type parameter we can use any valid java identifier but it convention to
use T always.
            eg: class Test<T>{}
                   class Test<iNeuron>{}

=> We can pass any no of type parameters need not be one.
            eg: class HashMap<K,V>{}
      HashMap<Integer,String> h=new HashMap<Integer,String>();

Which of the following are valid?
class Test <T extends Number&Runnable> {}//valid
      Number -> class
      Runnable-> interface

class Test<T extends Number&Runnable&Comparable> {} //valid
      Number -> class
      Runnable-> interface
      Comparable -> interface

class Test<T extends Number&String> {} //invalid
      we can't extends more than one class at a time.

class Test<T extends Runnable&Comparable> {} //valid
      Runnable-> interface
      Comparable -> interface

class Test<T extends Runnable&Number> {}//invalid
      Runnable-> interface
      Number -> class
            rule: first inherit and the implement so invalid

GenericClass
==========
      class: Type parameter

Can we apply TypeParameter at MethodLevel?
      Ans.Yes, it is possible.

Generic methods and wild-card character (?)
=====================================
1. methodOne(ArrayList<String>al):
      This method is applicable for ArrayList of only String type.

methodOne(ArrayList<String> al){
      al.add("sachin");
      al.add("navinreddy");
      al.add("iNeuron");
      al.add(new Integer(10));//invalid
}
Within the method we can add only String type of objects and null to the List.

2.  methodOne(new ArrayList<String>());
     methodOne(new ArrayList<Integer>());
     methodOne(new ArrayList<Runnable>());
                              |
                              |ArrayList<?> l =new ArrayList<String>();
```

```
                              |
      methodOne(ArrayList<?> l):
            We can use this method for ArrayList of any type but within the method
we can't add anything to the List except null.
Example:
      l.add(null);//(valid)
      l.add("A");//(invalid)
      l.add(10);//(invalid)
This method is useful whenever we are performing only read operation.

3. methodOne(ArrayList<? extends X> al)
      X -> class, we can make a call to method by passing ArrayList of X type or
its Child type.
      X -> interface, we can make a call to method by passing ArrayList of X type
or its Implementation class.

methodOne(ArrayList<? extends X> al){
      al.add(null);
}
      Best suited only for read operation.

4. methodOne(ArrayList<? super X> al)
      X -> class, we can make a call to method by passing ArrayList of X type or
its super class
      X -> interface, we can make a call to method by passing ArrayList of X type
or its super class of implementation class of x.

methodOne(ArrayList<? super X> al){
      al.add(X);
      al.add(null);
}


Which of the following declarations are allowed?

1. ArrayList<String> l1=new ArrayList<String>();//valid

2. ArrayList<?> l2=new ArrayList<String>();//valid

3. ArrayList<?> l3=new ArrayList<Integer>();//valid

4. ArrayList<? extends Number> l4=new ArrayList<Integer>();//valid

5. ArrayList<? extends Number> l5=new ArrayList<String>();//invalid

6. ArrayList<?> l6=new ArrayList<? extends Number>(); //invalid

7. ArrayList<?> l7=new ArrayList<?>(); //invalid


TypeParameter at Method level
==========================

                    |=> TypeParameter at the class level
class Demo<T>{

                          |=> Type parameter defined just before the return type
            public   <T>   void m1(T t){
```

```
            }
}
```

```
Which of the following declarations are allowed?
public <T> void methodOne1(T t){}//valid
public<T extends Number> void methodOne2(T t){}//valid
public<T extends Number&Comparable> void methodOne3(T t){}//valid
public<T extends Number&Comparable&Runnable> void methodOne4(T t){}//valid
public<T extends Number&Thread> void methodOne(T t){}//invalid
public<T extends Runnable&Number> void methodOne(T t){}//invalid
public<T extends Number&Runnable> void methodOne(T t){}//valid
```

```
Communication with non generic code
===============================
To provide compatibility with old version sun people compramized the concept of
generics in very few area's the
following is one such area.
```

```
Example:
importjava.util.*;
class Test{
      public static void main(String[] args){

            ArrayList<String> l=new ArrayList<String>();
            l.add("sachin");
            //l.add(10);//C.E:cannot find symbol,method add(int)

            methodOne(l);
            l.add(10.5);//C.E:cannot find symbol,method
                              add(double)

            System.out.println(l);//[sachin, 10, dhoni, true]
}
public static void methodOne(ArrayList l){
            l.add(10);
            l.add("dhoni");
            l.add(true);
}
```

```
Conclusions :
Generics concept is applicable only at compile time, at runtime there is no such
type of concept.
At the time of compilation, as the last step generics concept is removed,hence for
jvm generics syntax won't be available.
Hence the following declarations are equal.

      ArrayList l=new ArrayList<String>();
      ArrayList l=new ArrayList<Integer>();
      ArrayList l =new Arraylist<Double>();

All are equal at runtime,becoz compiler will remove these generics syntax
      ArrayList l=new ArrayList();
```

```
Example 1:
import  java.util.*;
class Test {
      public static void main(String[] args) {
```

```
            ArrayList l=new ArrayList<String>();
            l.add(10);
            l.add(10.5);
            l.add(true);
            System.out.println(l);// [10, 10.5, true]
      }
}
```

Example 2:

```
import java.util.*;
class Test {
      public void methodOne(ArrayList<String> l){}
      public void methodOne(ArrayList<Integer> l){}
}
```

CE: duplicate methods found

Behind the scenes by the compiler

==========

1. Compiler will scan the code
2. Check the argument type
3. if Generics found in the argument type remove the Generics syntax
4. Compiler will again check the syntax

Example3:

The following 2 declarations are equal.

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

For these ArrayList objects we can add only String type of objects.

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

Comparable vs Comparator

==========

```
public TreeSet();
```

            |=> When we use the above constructor,JVM will internally use Comparable interface method to sort the Objects
                  based on default natural sorting order.

What is Comparable interface?

      It is a functiaonal interace present in java.lang package.
      This interface is internally used by TreeSet object during sorting process of the Object.

```
  @FunctionalInterface
  public interface java.lang.Comparable<T> {
      public abstract int compareTo(T);
  }
```

eg#1.

```
import java.util.*;

class Test
{
      public static void main(String[] args)
      {
            //Sorting of objects will happen based on default natural sorting order
            TreeSet ts = new TreeSet();
```

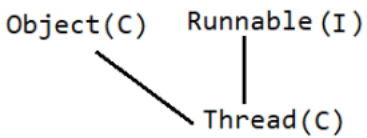
```
                  ts.add("A");
                  ts.add("Z");
                  ts.add("L");
                  ts.add("B");
                  ts.add(null);//NullPointerException
                  ts.add(10);//ClassCastException

                  System.out.println(ts);//[A,B,L,Z]
      }
}

Note:
      If we are keeping the data inside TreeSet object, then the data should be
                  a. Homogenous ====> because it uses compareTo() to sort the
Object
                  b. The object should compulsorily implements an interface called
"Comparable".
                        if we fail to do so , it would result in
"ClassCastException".

eg#2.
import java.util.*;
class Test
{
      public static void main(String[] args)
      {
            //Sorting of objects will happen based on default natural sorting order
            TreeSet ts = new TreeSet();

            ts.add(new StringBuffer("A"));
            ts.add(new StringBuffer("Z"));
            ts.add(new StringBuffer("L"));
            ts.add(new StringBuffer("B"));

            System.out.println(ts);//ClassCastException

      }

}
note: All Wraper classes and String class has implemented "Comparable" interface.
           StringBuffere class has not implemented Comparable interface, so the
above program would
         result in "ClassCastException".
```

```
ArrayList<String> al =new ArrayList<String>();
          al.add("sachin");
          al.add("dhoni");
          ;;;;;

          m1(al);

ArrayList<Integer> al1 =new ArrayList<Integer>();
        al1.add(10);
        al1.add(20);
          ;;;;;
         m1(al1);
```

GenericMethod with wild card pattern

```
void m1(ArrayList< ?    > al){
                    String Integer
    al.add("sachin");
    al.add("navinreddy");
    al.add("hyder");

    al.add(new Integer(10)); //invalid

}
```

1. m1(ArrayList<String> al)
2. m1(ArrayList<?> al)
3. m1(ArrayList<? extends X> al)
4. m1(ArrayList<? super X> al)



Functional interfaces
=================
=> If an interface contains only one abstract method then such interfaces are called as "Functional interface".

```
public interface java.util.function.Predicate<T> {
      public abstract boolean test(T);

      //default methods
      public java.util.function.Predicate<T> and(java.util.function.Predicate<?
super T>);
      public java.util.function.Predicate<T> negate();
      public java.util.function.Predicate<T> or(java.util.function.Predicate<?
super T>);

      //static method
      public static <T> java.util.function.Predicate<T> isEqual(java.lang.Object);
}
```

Usage of Predicate
===============

```
class MyPredicate implements Predicate<Integer>
{
      @Override
      public boolean test(Integer i){
            if (i>10)
                  return true;
            else
                  return false;
      }
}
```

Instead of writing a seperate class, we can write lambda expression as shown below
==========

```
class Test {
      public static void main(String[] args){
            Predicate<Integer> p =i->i>10
            System.out.println(p.test(10));//false
            System.out.println(p.test(100));//true
      }
}
```

Write a Predicate to check whether the given String length is >=3 or not?

Instead of writing a seperate class, we can write lambda expression as shown below
==========

```
class MyPredicate implements Predicate<String>
{
      @Override
      public boolean test(String name){
                  if(name.length()>=3)
                        return true;
                  else
                        return false;
      }
}

class Test {
      public static void main(String[] args){
```

```
            Predicate<String> p = name -> name.length() >= 3;
            System.out.println(p.test("PWC"));//true
            System.out.println(p.test("CS"));//false
      }
}
```

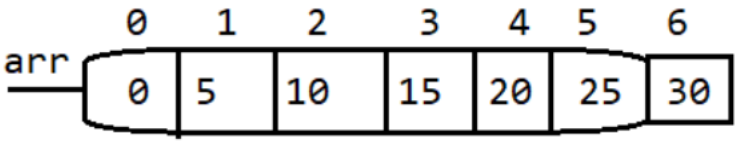
default methods available as utility methods for developer

```
==============================================
  public default  Predicate<T> and(Predicate p);
  public defualt  Predicate<T> negate();
  public default  Predicate<T> or(Predicate p);

import java.util.function.*;

public class Test {
      public static void main(String[] args){

            int[] arr = {0,5,10,15,20,25,30};

            Predicate<Integer> p1 = i-> i>10;
            System.out.println("Elements greater than 10 are :: ");
            m1(p1,arr);

            Predicate<Integer> p2 = i-> i%2==0;
            System.out.println("Elements which are even no :: ");
            m1(p2,arr);


            System.out.println("Eleemnts which are greater than 10 and should be
even no");
            m1(p1.and(p2),arr);

            System.out.println("Eleemnts which are greater than 10 or should be
even no");
            m1(p1.or(p2),arr);

            System.out.println("The elements which are not even are :: ");
            m1(p2.negate(),arr);
      }

      public static void m1(Predicate<Integer> p , int[] x){
            for (int ele: x )
                  if (p.test(ele))//ele-> ele>10
                        System.out.println(ele);

      }
}

Function(I)
=========
T->  input type
R-> return type

public interface java.util.function.Function<T, R> {

 // 1 abstract method
  public abstract R apply(T);

  //default methods
```

```
  public <V> java.util.function.Function<V, R>
compose(java.util.function.Function<? super V, ? extends T>);
  public <V> java.util.function.Function<T, V>
andThen(java.util.function.Function<? super R, ? extends V>);

  //static method
  public static <T> java.util.function.Function<T, T> identity();
}

Writing a code using Implementation class
==================================
class MyFunction implements Function<String,Integer>
{
      @Override
      public Integer apply(String name){
                  return name.length();
      }
}

public class Test {
      public static void main(String[] args){
            Function<String,Integer> f = new MyFunction();
            int output = f.apply("sachin");
            System.out.println(output);

            System.out.println("sachin".length());
      }
}

Coding using LambdaExpression
============================
public class Test {
      public static void main(String[] args){
            Function<String,Integer> f = name -> name.length();
            int output = f.apply("sachin");
            System.out.println(output);
      }
}

Note:
  When to go for Predicate and When to go for Function?
      Predicate -> To implement some conditional checks we should go for Predicate
      Function ->  To perform some operation and to return some result we should go
for Function.


MethodReference(::) and Consturctor reference(::)
=========================================
  ::  ====> Scope resolution operator

syntax for method reference
=======================
1. static method
            ClassName::methodName

2. instance method
             object:: methodName

eg#1.
```

```
public class Test {
	public static void m1(){
			for (int i = 1;i<=10 ;i++ )
			{
				System.out.println("child thread");
			}
	}
	public static void main(String[] args) throws Exception{

		//using method reference binded the method call of run() of interface
Runnable
		Runnable r = Test::m1;

		Thread t =new Thread(r);
		t.start();

		for (int i = 1;i<=10 ;i++ )
		{
			System.out.println("main thread");
			Thread.sleep(1000);
		}

	}
}
```

```
int[] arr = {0,5,10,15,20,25,30};
```



m1(p1,arr)

Predicate<Integer> p1

elements greater than 10 — 1st case

i -> i > 10

m1(p2,arr)

Predicate<Integer> p2

elements which are even in number — 2nd case

i -> i%2 == 0

```
public static void m1(Predicate<Integer> p, int[] x){

   for ( int ele : x)
       if(p.test(ele))
           System.out.println(ele)

 }
```

```
class Sample
{
	private String s;
	Sample(String s){
		this.s = s;
		System.out.println("Constructor executed...."+s);
	}
}
@FunctionalInterface
interface Interf
{
	public Sample get(String s);
}
public class Test {
	public static void main(String[] args){
			Interf i  = s -> new Sample(s);
			i.get("from lambda expression...");

			System.out.println();

			//constructor reference
			Interf i1 = Sample::new;
			i1.get("from constructor reference....");
	}
}
```

Example of Method reference

```
@FunctionalInterface
interface Interf
{
	public void m1(int i);
}
public class Test {

	//logic coded by other developer
	public void m2(int i){
		System.out.println(i*i);
		System.out.println("logic coming from method reference...");
	}

	public static void main(String[] args){
			Interf i  = x-> System.out.println(x);
			i.m1(10);

			System.out.println();

			//method reference(binding the body of m2() to abstract method
m1)
			Interf i1 = new Test()::m2;
			i1.m1(20);

	}
}
```

Eg:To demonstrate the usage of forEach() to print the elements of Arraylist

```
import java.util.*;
import java.util.function.*;
```

```
// public void forEach(java.util.function.Consumer<? super E>);
// public abstract void accept(T t)

class MyConsumer implements Consumer<String>
{
      @Override
      public void accept(String name){
            System.out.println("accept method got called...");
            System.out.println(name);
      }
}
public class Test {
      public static void main(String[] args){

                  ArrayList<String> names = new ArrayList<String>();
                  names.add("sachin");
                  names.add("dhoni");
                  names.add("kohli");
                  names.add("dravid");

                  //Traditional approach
                  Consumer<String> consumer = new MyConsumer();
                  names.forEach(consumer);
                  System.out.println();

                  //lambda expression
                  names.forEach(name->System.out.println(name));
                  System.out.println();

                  //method reference
                  names.forEach(System.out::println);
      }
}
```

Stream API
==========
Stream -----> Channel through which there is a free flow movement of data.

Streams
To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between Java.util.streams and Java.io streams?
java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection
but Java.io streams meant for processing binary and character data with respect to file.
i.e it represents stream of binary data or character data from the file .
hence Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?
 => If we want to represent a group of individual objects as a single entity then We should go for collection.
 => If we want to process a group of objects from the collection then we should go for streams.
 => We can create a stream object to the collection by using stream() method of Collection interface. stream()
       method is a default method added to the Collection in 1.8 version.

```
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args){

		ArrayList<Integer> al = new ArrayList<Integer>();
			al.add(0);
			al.add(5);
			al.add(10);
			al.add(15);
			al.add(20);
			al.add(25);

			System.out.println(al);//[0,5,10,15,20,25]

			//till jdk1.7v
			ArrayList<Integer> evenList = new ArrayList<Integer>();
			for ( Integer i1: al )
				if (i1%2==0)
					evenList.add(i1);
			System.out.println(evenList);//[0,10,20]

			//From JDK1.8V we use Streams
			//1. Configuration ===> al.stream()
			//2. Processing       ===> filter(i->i
%2==0).collect(Collectors.toList())
			List<Integer> streamList=al.stream().filter(i->i
%2==0).collect(Collectors.toList());
			System.out.println(streamList);
			streamList.forEach(System.out :: println);

	}
}

eg#2.
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args){
		ArrayList<Integer> al = new ArrayList<Integer>();
			al.add(0);
			al.add(5);
			al.add(10);
			al.add(15);
			al.add(20);
			al.add(25);

			System.out.println(al);

			// till JDK1.7V
			ArrayList<Integer> doubleList = new ArrayList<Integer>();
			for ( Integer i1: al )
				doubleList.add(i1*2);

			System.out.println(doubleList);


			// from JDK1.8V
			// map-> for every object, if a new object has to be created then
```

```
go for Map
                  List<Integer> streamList = al.stream().map(obj-
>obj*2).collect(Collectors.toList());
                  System.out.println(streamList);
                  streamList.forEach(i-> System.out.println(i));

                  System.out.println();

                  streamList.forEach(System.out::println);
      }
}
```
=> Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.

We can process the objects in the following 2 phases

1.Configuration

2.Processing

1) Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering:

We can configure a filter to filter elements from the collection based on some boolean condition by using

filter()method of Stream interface.

```
                  public Stream filter(Predicate<T> t)
```
here (Predicate<T > t) can be a boolean valued function/lambda expression

Ex:
```
Stream s = c.stream();
Stream s1 = s.filter(i -> i%2==0);
```
Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for

map() method of Stream interface.

```
                  public Stream map (Function f);
```
It can be lambda expression also

Ex:
```
 Stream s = c.stream();
 Stream s1 = s.map(i-> i+10);
```
Once we performed configuration we can process objects by using several methods.

2) Processing

processing by collect() method

Processing by count()method

Processing by sorted()method

Processing by min() and max() methods

forEach() method

toArray() method

Stream.of()method

eg#1.
```
import java.util.*;
import java.util.stream.*;
public class Test {
      public static void main(String[] args){
```

```
            ArrayList<String>names = new ArrayList<String>();
            names.add("sachin");
            names.add("saurav");
            names.add("dhoni");
            names.add("dravid");
            names.add("kohli");
            names.add("raina");

            System.out.println(names);

            List<String> reslut = names.stream().filter(name-
>name.length()>5).collect(Collectors.toList());
            System.out.println(reslut.size());

            long count= names.stream().filter(name->name.length()>5).count();
            System.out.println("The no of objects whose string length > 5
is ::"+count);

      }
}

import java.util.*;
import java.util.stream.*;

//Comparable(Predefined API for natural sorting order) -> compareTo(Object obj)
//Comparator(for userdefined class for customized sorting order)->
compare(Obj1,Obj2)

public class Test {
      public static void main(String[] args){

            ArrayList<Integer> al =new ArrayList<Integer>();
                  al.add(10);
                  al.add(0);
                  al.add(15);
                  al.add(5);
                  al.add(20);
            System.out.println("Before sorting :: "+al);

            //using stream api
            List<Integer> resultList=
al.stream().sorted().collect(Collectors.toList());
            System.out.println("After sorting  :: "+resultList);

            List<Integer> customizedResult = al.stream().sorted((i1,i2)-
>i2.compareTo(i1)).collect(Collectors.toList());
            System.out.println("After sorting  :: "+customizedResult);

      }
}
import java.util.*;
import java.util.stream.*;

//Comparable(Predefined API for natural sorting order) -> compareTo(Object obj)
//Comparator(for userdefined class for customized sorting order)->
compare(Obj1,Obj2)

public class Test {
```

```
	public static void main(String[] args){

		ArrayList<Integer> al =new ArrayList<Integer>();
			al.add(10);
			al.add(0);
			al.add(15);
			al.add(5);
			al.add(20);
		System.out.println("Array List is ::"+al);

		Object[] objArr = al.stream().toArray();
		for(Object obj: objArr)
			System.out.println(obj);

		System.out.println();

		Integer[] objArr1 = al.stream().toArray(Integer[]::new);
		for(Integer obj1: objArr1)
			System.out.println(obj1);


	}
}

eg
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args){

		//Stream API ===> Collections(group of objects)

		Stream s= Stream.of(9,99,999,9999,99999);
		s.forEach(System.out::println);

		System.out.println();

		Double[] d = {10.0,10.1,10.2,10.3,10.4};
		Stream s1= Stream.of(d);
		s1.forEach(System.out::println);

	}
}
collect()
=======
This method collects the elements from the stream and adding to the specified to
the collection indicated (specified) by argument.


eg#1.
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args) {

			ArrayList<String> names = new ArrayList<String>();
			names.add("sachin");
			names.add("saurav");
			names.add("dhoni");
```

```
                  names.add("yuvi");


                  System.out.println(names);//[sachin,saurav,dhoni,yuvi]


      //Predicate(I)

//    public abstract boolean test(T);
                  List<String> result=names.stream().filter(name->name.length()>5).
                        collect(Collectors.toList());
                  System.out.println(result);


//Function(I)<T,R>

      //    public abstract R apply(T);
                  List<String> mapResult = names.stream().map(name->
name.toUpperCase()).
                                    collect(Collectors.toList());
                  System.out.println(mapResult);

      }
}


count()
======
This method returns number of elements present in the stream.
      public long count()

import java.util.*;
import java.util.stream.*;
public class Test {
      public static void main(String[] args) {

                  ArrayList<String> names = new ArrayList<String>();
                  names.add("sachin");
                  names.add("saurav");
                  names.add("dhoni");
                  names.add("yuvi");


                  System.out.println(names);//[sachin,saurav,dhoni,yuvi]

                  long count = names.stream().filter(name-
>name.length()>5).count();
                  System.out.println(count);
      }
}

III.Processing by sorted()method
     If we sort the elements present inside stream then we should go for sorted()
method.
     The sorting can either default natural sorting order or customized sorting
order specified by comparator.
            sorted()- default natural sorting order
            sorted(Comparator c)-customized sorting order.
```

```
import java.util.*;
import java.util.stream.*;

public class Test {
      public static void main(String[] args) {

                  ArrayList<Integer> al = new ArrayList<Integer>();
                  al.add(10);
                  al.add(20);
                  al.add(0);
                  al.add(5);
                  al.add(25);
                  al.add(15);
                  System.out.println(al);

                  List<Integer> result =
al.stream().sorted().collect(Collectors.toList());
                  System.out.println(result);

                  List<Integer> customizedResult=al.stream().sorted((i1,i2) ->-
i1.compareTo(i2)).collect(Collectors.toList());
                  System.out.println(customizedResult);
      }
}
```

IV.Processing by min() and max() methods

```
            min(Comparator c)
                        returns minimum value according to specified comparator.
            max(Comparator c)
                        returns maximum value according to specified comparator.
```

```
import java.util.*;
import java.util.stream.*;

public class Test {
      public static void main(String[] args) {

                  ArrayList<Integer> al = new ArrayList<Integer>();
                  al.add(10);
                  al.add(20);
                  al.add(0);
                  al.add(5);
                  al.add(25);
                  al.add(15);
                  System.out.println(al);

                  Integer minValue = al.stream().min((i1,i2)->
i1.compareTo(i2)).get();
                  System.out.println(minValue);

                  Integer maxValue = al.stream().max((i1,i2)->
i1.compareTo(i2)).get();
                  System.out.println(maxValue);
      }
}
```

V.forEach() method

This method will not return anything.

This method will take lambda expression as argument and apply that lambda

expression for each element present in the stream.

```
import java.util.*;
import java.util.stream.*;

public class Test {
      public static void main(String[] args) {

                  ArrayList<String>names = new ArrayList<String>();
                  names.add("AAA");
                  names.add("BBB");
                  names.add("CCC");
                  names.add("DDD");

                  names.stream().forEach(name -> System.out.println(name));
                  names.stream().forEach(System.out::println);
      }
}
```

VI.
toArray() method
We can use toArray() method to copy elements present in the stream into specified array

```
import java.util.*;
import java.util.stream.*;

public class Test {
      public static void main(String[] args) {
                  ArrayList<Integer> al =new ArrayList<Integer>();
                  al.add(0);
                  al.add(10);
                  al.add(5);
                  al.add(20);
                  al.add(15);
                  System.out.println(al);

                  Integer[] array = al.stream().toArray(Integer[]::new);
                  for (Integer element : array)
                        System.out.println(element);
      }
}
```

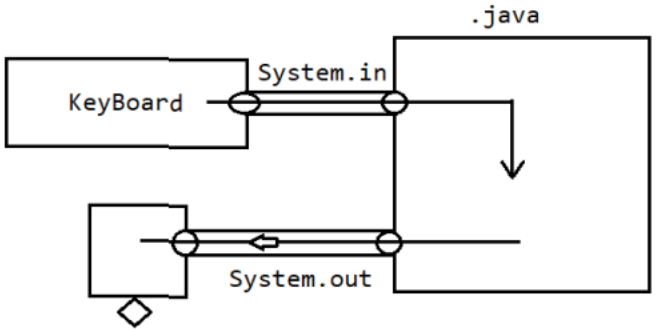
VII.Stream.of()method
We can also apply a stream for group of values and for arrays.

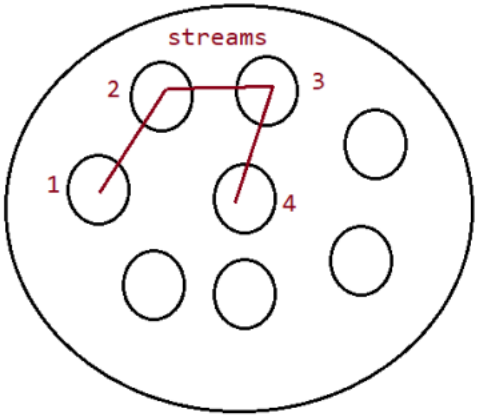
Ex:

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```

(java.io.*)

Input and Output operation





1,2,3,4-> objects choosen for processing then go for "Stream".

1. Configuration
2. Processing

Collection(large volume of data)

To process this Collection Objects also we use "Stream"

(java.util.stream.*)