

Collection

Wrapper class:

A wrapper Class is used to convert the primitive to non-primitive and non-primitive back to the primitive.

Boxing

- The process of wrapping the primitive type data to non-primitive type data is known as boxing.
- from 1.8 version of JDK, boxing can be performed automatically. Hence it is also known as Autoboxing.

Unboxing

- The process of unwrapping non-primitive type data that is wrapper object back to the primitive is known as unboxing.
- From 1.8 version of JDK unboxing can be performed automatically. Hence it is known as Auto unboxing.
- To perform the boxing and unboxing for the add primitive data types dedicated classes are created lead classes are known as wrapper class.

byte	->	Byte
short	->	Short
int	->	Int
long	->	Long
float	->	Float
double	->	Double
char	->	Char
boolean	->	Boolean

Collection framework

It contains set of classes and interfaces. That is used to store multiple objects together.

Characteristics

- In collection framework only non-primitive values are allowed. It is used to store the heterogeneous object together
- The size is not fixed.
- Inserting and removing the multiple objects is possible
- In collection framework. **CRUD** Operation can be performed.

Root interfaces of collection:

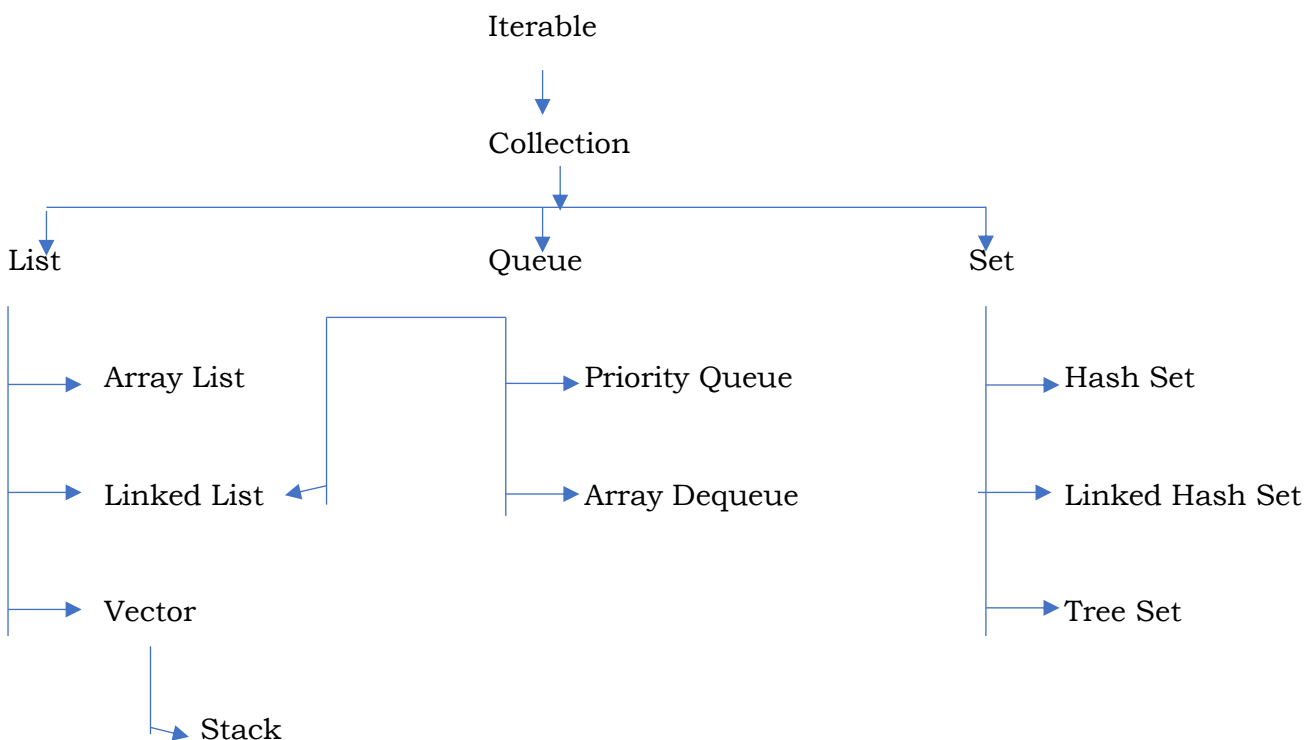
Collection has two root interfaces

Java.util.Collection

Java.util. Map

Java.util.Collection

- It is an interface defined in java.util.package
- It is used to store multiple objects
- List, Queue and set are the sub interfaces of collection
- By using collection interface CRUD operations are performed on objects



Abstract methods of java.util.Collection

Function	Method Name	Return type
ADDING	add(Object o)	boolean
	addAll(Collection c)	boolean
REMOVING	remove(Object o)	boolean
	removeAll(Collection c)	boolean
	retainAll(Collection c)	boolean
	clear	void
ACCESSING	iterator()	iterator
SEARCHING	contains(Object o)	boolean
	containsAll(Collection c)	boolean
OTHERS	size()	int
	isEmpty()	boolean
	toArray()	Object[]
	hashCode()	int
	toString()	string
	equals(Object o)	boolean

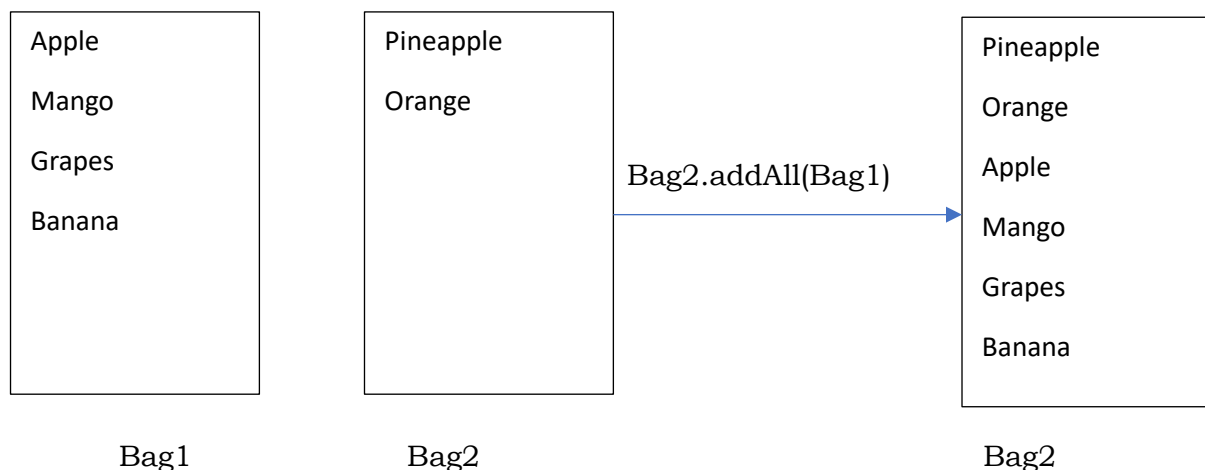
➤ **public abstract boolean add(Object o):**

It is used to add an object to the collection

➤ **public abstract boolean addAll(Collection c):**

It is used to add a group of objects from one collection to another collection

Example:

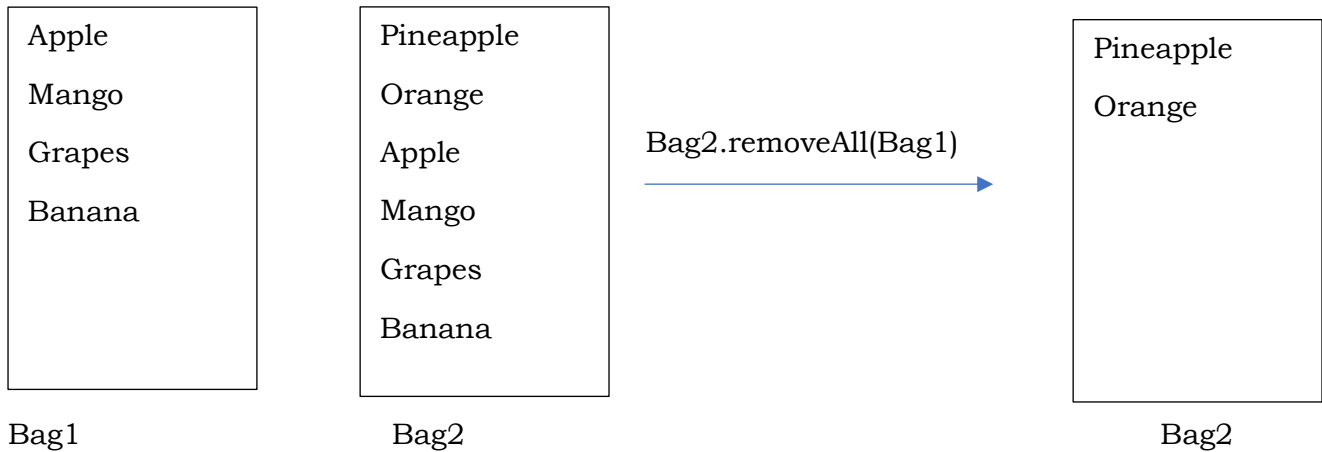


➤ **public abstract boolean remove(Object o):**

It is used to remove an object from the collection

➤ **public abstract boolean removeAll(Collection c):**

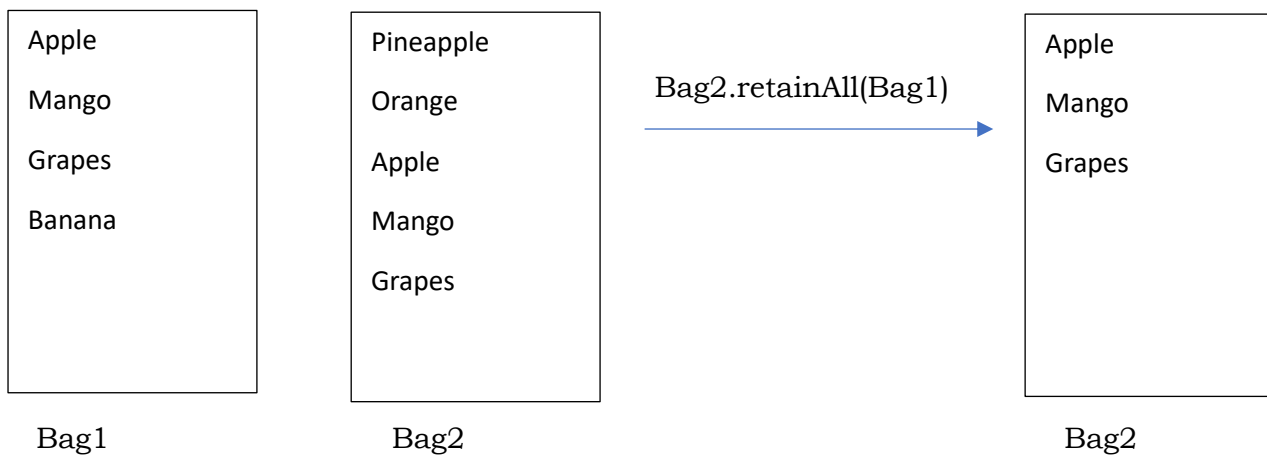
It is used to remove all the objects from one collection that are available in another collection



➤ **Public abstract Boolean retainAll(Collection c)**

It is used to retain all the objects in one collection that are available in another collection

Other than the retained objects, remaining objects are removed



➤ **Public abstract void clear();**

It is used to clear all the object in the collection

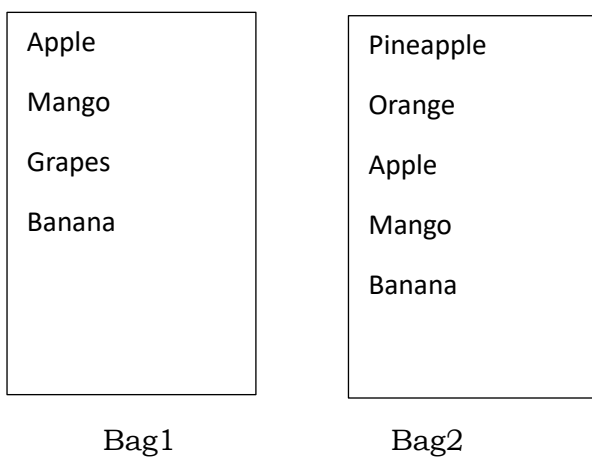
➤ **Public abstract Boolean contains(Objects o)**

It is used to search an object in the collection

If the collection contains an object, it returns true else it returns false

➤ **Public abstract Boolean containsAll(Objects o)**

If one collection contains all the objects of another collection if return true else it returns false



Bag2.containsAll(Bag2); // false

// banana is not available in Bag2

▪ List

Java.util.List;

- It is the sub interface of java.util.Collection
- List contains the methods inherited from the collection and also its own methods
- It is introduced in 1.2 version of JDK.

Characteristics:

NIDHI

- 1) Any number of nulls can be inserted
- 2) In a list insertion order of an object is maintained
- 3) Duplicate objects are allowed
- 4) Heterogeneous objects can be inserted
- 5) Objects can be accessed by using an Index

Abstract methods of List Package

List contains the methods of collection interface and also the mentioned methods

Function	Method Name	Return type
ADDING	add(int index, object o)	void
	addAll(int index, Collection c)	boolean
REMOVE	remove(int index)	boolean
ACCESS	get(int index)	object
	listIterator()	ListIterator
	listIterator(int index)	ListIterator
SEARCH	indexOf(Object)	int
	lastIndexOf(object)	int

➤ public abstract void add(int index, Object) :

It is used to add an object at the specified index.

➤ public abstract void addAll(int index, Collection) :

It is used to add all the objects from one Collection to another collection from the specified index.

➤ **public abstract Object get(int index) :**

it is used to return the object from the specified index.

➤ **Public abstract int indexOf(Object) :**

It is used to return the index of the object that has the first occurrence

➤ **public abstract int LastIndexOf(Object) :**

it is used to return the index of an object that has the last occurrence

If the object is not available in the collection both **indexOf()** and **LastIndexOf()** method return **-1**.

ArrayList

Java.util.ArrayList :

It is the subclass of java.util.List interface.

It is used to store multiple objects in the sequential order

It is implemented by using growable Array / Resizable Array data structure



Characteristics:

NIDHI

- 1) Any number of nulls can be inserted
- 2) In a list insertion order of an object is maintained
- 3) Duplicate objects are allowed
- 4) Heterogeneous objects can be inserted
- 5) Objects can be accessed by using an Index

Constructors of Array List:

- 1) **ArrayList() :** It is used to create one **empty ArrayList** with initial capacity of 10.

ArrayList al = new ArrayList();

--	--	--	--	--	--	--	--	--	--

- 2) **ArrayList(Collection c) :** it is used to create a new ArrayList that is initialized with the objects in the specified Collection.

Al ->

10	20	30	40	50
----	----	----	----	----

ArrayList al2 = new ArrayList(al1);

3) ArrayList (int initialCapacity) : it is used to create a new empty ArrayList object with specified initial capacity

ArrayList al = new ArrayList(5);

Al ->

--	--	--	--	--

Example :

```
import java.util.ArrayList;

public class Store {
    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        System.out.println(al); // al.toString() --> [10,20,30,40,50]
    }
}
```

10	20	30	40	50					
----	----	----	----	----	--	--	--	--	--

o/p :

```
Note: Recompile with -Xlint:unchecked for details
PS C:\Users\Saurabh\Desktop\java folder> java S
[10, 20, 30, 40, 50]
PS C:\Users\Saurabh\Desktop\java folder> 
```

Example

```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();
        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        System.out.println(al1);

        ArrayList al2 = new ArrayList<>();
        al2.add("car");
        al2.add("Cycle");
        al2.add("UFO");

        System.out.println(al1.contains("Bike"));
        System.out.println(al1.containsAll(al2));
        System.out.println(al1.indexOf("Bus"));
        System.out.println(al1.lastIndexOf("Bus"));

    }
}
```

o/p:

```
PS C:\Users\Saurabh\Desktop
[Car, Bus, Bike, Cycle, Bus]
true
false
1
4
Live Share  Java: Ready
```

Example

```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();

        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        ArrayList al2 = new ArrayList<>();

        al2.add("Car");
        al2.add("Cycle");
        al2.add("UFO");
        al2.add("Rocket");

        al2.retainAll(al1);
        System.out.println(al2);

    }
}
```

o/p :

```
PS C:\Users\Saurabh\Desktop\java folder> java Remove
Note: Remove.java uses unchecked or unsafe op
Note: Recompile with -Xlint:unchecked for det
PS C:\Users\Saurabh\Desktop\java folder> java Remove
[Car, Cycle, UFO]
PS C:\Users\Saurabh\Desktop\java folder>
```

```
import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();

        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

        System.out.println(al1.get(0));
        System.out.println(al1.get(1));
        System.out.println(al1.get(2));
        System.out.println(al1.get(3));
        System.out.println(al1.get(4));
        System.out.println(al1.get(5));

    }
}
```

```

import java.util.ArrayList;

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

        ArrayList al1 = new ArrayList<>();

        al1.add("Car");
        al1.add("Bus");
        al1.add("Bike");
        al1.add("Cycle");
        al1.add("Bus");
        al1.add("UFO");

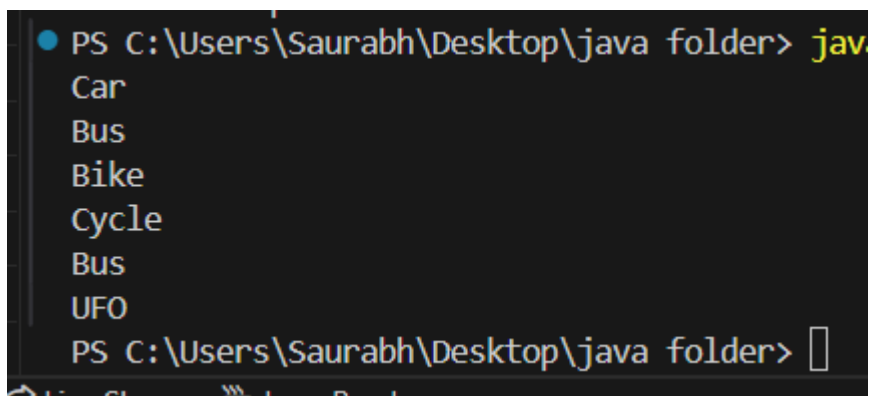
        int size = al1.size();

        for (int i = 0; i < size; i++) {
            System.out.println(al1.get(i));
        }

    }
}

```

O/P :



```

PS C:\Users\Saurabh\Desktop\java folder> java
Car
Bus
Bike
Cycle
Bus
UFO
PS C:\Users\Saurabh\Desktop\java folder>

```

public Iterator iterator() :

- it is used to access the objects in the collection
- it creates java.util.Iterator type object and returns the address.
- It is used to traverse or iterate the objects in the collection
- It is declared in java.util.Iterable Interface

➤ **Java.util.Iterator**

- It is an interface that is defined in java.util package
- It is used to Iterate or Traverse the objects on the collection

Abstract methods of Iterator:

➤ **Public abstract Object next():**

It is used to return the object that is present next to the cursor and it moves the cursor to the next position

➤ **Public abstract Boolean hasNext() :**

- It is used to check whether any element is present next to the cursor or not
- If the object is available next to the cursor, it returns true else, it returns false

➤ **Public abstract Boolean remove() :**

It is used to remove the object that is recently/ currently iterated by next() method.

Note :

During Iteration if we try to modify the actual collection by using collection method then we get concurrent modification exception

- **ConcurrentModificationException**

If the remove method is called without calling the next method, then we get illegal state exception

- **IllegalStateException**

Access :

```
import java.util.ArrayList;
import java.util.*;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        Iterator i = al.iterator();
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
    }
}
```

```
import java.util.ArrayList;
import java.util.*;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        Iterator i = al.iterator();
        while (i.hasNext()){
            System.out.println(i.next());
        }
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java A
10
20
30
40
50
```

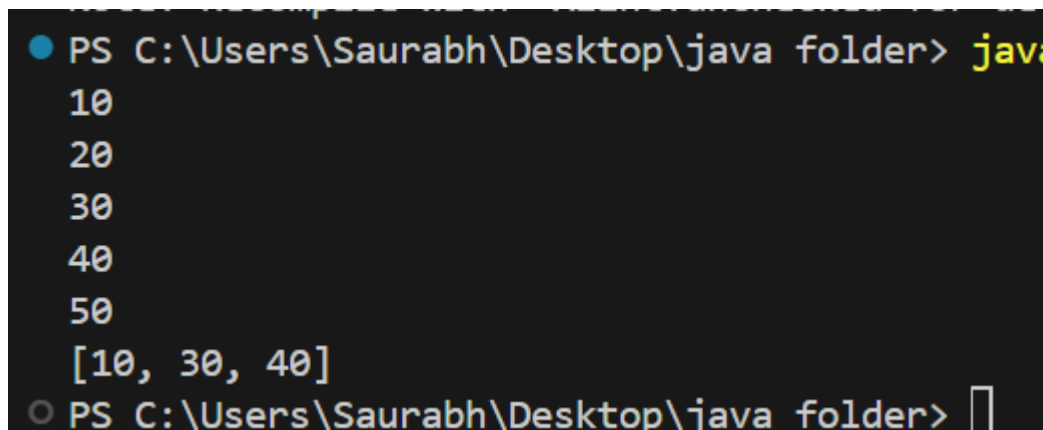
Remove :

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

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        Iterator i = al.iterator();
        System.out.println(i.next());
        System.out.println(i.next());
        i.remove();
        System.out.println(i.next());
        System.out.println(i.next());
        System.out.println(i.next());
        i.remove();
        System.out.println(al);
    }
}
```



```
PS C:\Users\Saurabh\Desktop\java folder> java Remove
10
20
30
40
50
[10, 30, 40]
PS C:\Users\Saurabh\Desktop\java folder>
```

Limitations of java.util.Iterator;

- By using Iterator object iteration/traversing can be done only one time.
- Traversing in backward direction is not possible.
- During iteration adding and replacing the object is not possible.
- To overcome these limitations java.util.ListIterator can be used

➤ **ListIterator**

➤ **Public ListIterator listIterator();**

- It is used to create object of ListIterator type.
- It is declared in java.util.List Interface.
- It places the cursor in 0th position In ListIterator object

➤ **Public ListIterator listIterator(int position);**

- It is used to create object of ListIterator type
- It is declared in java.util.List Interface.
- It places the cursor in specified position In ListIterator object.

Java.util.ListIterator

- It is sub interface of java.util.iterator
- It is used to traverse the list.

Characteristics :

- By using ListIterator traversing can be done in both direction
- By using ListIterator, any number we can traverse by moving the cursor in both the direction
- In ListIterator adding and replacing during Iteration is possible

Methods:

List Iterator contains the methods that are inherited from java.util.Iterator and also its own declared methods

Inherited methods

- Public abstract object next()
- Public abstract boolean hasNext()
- Public abstract void remove()

Declared methods

➤ **Public abstract object previous();**

It is used to return the object that is present previous to the cursor and it moves the cursor to the previous position

➤ **Public abstract boolean hasPrevious():**

It is used to check whether any object is present previous to the cursor or not

If the object is available previous to the cursor else it returns false

➤ **Public abstract void add(Object o):**

It is used to add the object during iteration in the cursor pointing position

➤ **Public abstract void set(Object o):**

It is used to replace the object that is currently iterated by next() or previous()

method.

Access:

```
import java.util.ArrayList;
import java.util.ListIterator;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        ListIterator li = al.listIterator();
        while (li.hasNext()) {
            System.out.println(li.next());
        }

        while (li.hasPrevious()) {
            System.out.println(li.previous());
        }
    }
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Access
10
20
30
40
50
+++++
50
40
30
20
10
PS C:\Users\Saurabh\Desktop\java folder>
```

With position

```
import java.util.ArrayList;
import java.util.ListIterator;

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

        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        ListIterator li = al.listIterator(4);
        System.out.println(li.next());
    }
}
```

```
● PS C:\Users\Saurabh\Desktop\java folder> java Access
50
PS C:\Users\Saurabh\Desktop\java folder> 
```

Replace:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Replace {
    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);

        ListIterator li = al.listIterator(3);
        System.out.println(li.previous());
        System.out.println(li.previous());
        System.out.println(li.previous());
        li.set(75);
        System.out.println(li.next());

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

```
PS C:\Users\Saurabh\Desktop\java folder> java Replace
30
20
10
75
[75, 20, 30, 40, 50]
PS C:\Users\Saurabh\Desktop\java folder> 
```

Example:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Access {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Saurabh");
        names.add("Ketan");
        names.add("Chetan");
        names.add("Omkar");
        names.add("Vikas");

        ListIterator<String> li = names.listIterator(names.size());
        while (li.hasPrevious()) {
            String name = li.previous();

            if (name.equals("Vikas")) {
                li.set("Vicky");
            }
        }

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

O/P :

```
● PS C:\Users\Saurabh\Desktop\java folder> javac Access.java
● PS C:\Users\Saurabh\Desktop\java folder> java Access
[Saurabh, Ketan, Chetan, Omkar, Vicky]
PS C:\Users\Saurabh\Desktop\java folder>
```

Example:

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Access {
    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(20);
        al.add(30);
        al.add(40);
        al.add(50);
        al.add(60);
        al.add(30);

        ListIterator li = al.listIterator();
        while (li.hasNext()) {
            if ((Integer) li.next() == 30) {
                li.set(60);
                break;
            }
        }
        System.out.println(al);
    }
}
```

O/P:

Note: Access.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

● PS C:\Users\Saurabh\Desktop\java folder> java Access

[20, 60, 40, 50, 60, 30]

PS C:\Users\Saurabh\Desktop\java folder> █

pwd-i-search: _

➤ For each loop:

For each loop/ advanced for loop is used only for array and collection

Syntax :

```
for ( datatype VarName : collection/Array){  
    // instructions  
}
```

- The type of variable declared in for each loop should be similar as type of collection or arrays
- The number of iterations of for each loop is always equals to the numbers of elements of array or collection

Workflow:

For every iteration one element is taken from the specified collection or array in a sequential order and store in the variable declared in for each loop.

Example:

```
import java.util.ArrayList;  
  
public class Access {  
    public static void main(String[] args) {  
        ArrayList<Integer> al = new ArrayList<>();  
        al.add(20);  
        al.add(30);  
        al.add(40);  
        al.add(50);  
        al.add(60);  
        al.add(30);  
  
        for (Object o : al) {  
            System.out.println(o);  
        }  
    }  
}
```

```
PS C:\Users\Saurabh\Desktop\java folder> java Access  
20  
30  
40  
50  
60  
30  
PS C:\Users\Saurabh\Desktop\java folder> 
```

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Access {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Saurabh");
        names.add("Ketan");
        names.add("Chetan");
        names.add("Omkar");
        names.add("Vikas");

        for (Object o : names) {
            System.out.println(o);
        }
    }
}
```

```
● PS C:\Users\Saurabh\Desktop\java folder> java Access
Saurabh
Ketan
Chetan
Omkar
Vikas
PS C:\Users\Saurabh\Desktop\java folder> 
```



```

import java.util.ArrayList;

public class Employeee {
    String name;
    int id;
    double salary;

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        for (Object o : al) {
            Employeee e = (Employeee) o;
            System.out.println(e.name);
            System.out.println(e.id);
            System.out.println(e.salary);
            System.out.println("-----");
        }
    }
}

```

```

PS C:\Users\Saurabh\Desktop\java folder> jav
Saurabh
1
50000.0
-----
vikas
2
49000.0
-----
Mayur
3
44000.0
-----
Omkar
4
45900.0
-----
PS C:\Users\Saurabh\Desktop\java folder> 

```

Write a java program to search employee based on ID.

Code:

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

public class Employeee {
    String name;
    int id;
    double salary;

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        int id = 1;
        boolean exist = false;

        Iterator i = al.iterator();
        while (i.hasNext()) {
            Employeee e = (Employeee) i.next();
            if (e.id == 1) {
                System.out.println("found");
                exist = true;
            }
        }

        if (exist != true) {
            System.out.println("not found");
        }
    }
}
```

```
PS C:\Users\Sau
found
PS C:\Users\Sau
```

Write a java program to print name and salary of employee based on employee id.

Code:

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

public class Employeee {
    String name;
    int id;
    double salary;

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        int id = 1;
        boolean exist = false;

        Iterator i = al.iterator();
        while (i.hasNext()) {
            Employeee e = (Employeee) i.next();
            if (e.id == 1) {
                System.out.println(e.name);
                System.out.println(e.salary);
                exist = true;
            }
        }

        if (exist != true) {
            System.out.println("not found");
        }
    }
}
```

```
● PS C:\Users\Saurabh>
Saurabh
50000.0
○ PS C:\Users\Saurabh>
```

Write a java program to increase salary of all the employees by 10%

```
package pack.subpack;

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

public class Employeee {

    String name;
    int id;
    double salary;

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" + "name='" + name + '\'' + ", id=" + id + ", salary=" +
salary + '}';
    }

    public void increaseSalary(int percentage) {
        this.salary += salary * percentage / 100;
    }

    public static void main(String[] args) {
        ArrayList<Employeee> al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));

        System.out.println("Salaries before increasing are : ");
        for (Employeee e : al) {
            System.out.println(e);
        }

        System.out.println("-----");
        for (Employeee e : al) {
            e.increaseSalary(10); // increasing the salaries by 10%
        }
    }
}
```

```
        System.out.println("Salaries after increasing are : ");
        for (Employee e : al) {
            System.out.println(e);
        }
    }
}
```

```
Salaries before increasing are :
Employee{name='Saurabh', id=1, salary=50000.0}
Employee{name='vikas', id=2, salary=49000.0}
Employee{name='Mayur', id=3, salary=44000.0}
Employee{name='Omkar', id=4, salary=45900.0}
-----
Salaries after increasing are :
Employee{name='Saurabh', id=1, salary=55000.0}
Employee{name='vikas', id=2, salary=53900.0}
Employee{name='Mayur', id=3, salary=48400.0}
Employee{name='Omkar', id=4, salary=50490.0}
```

Write a java program to remove the employee based on their Id

Ans:

```
package pack.subpack;

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

public class Employeee {

    String name;
    int id;
    double salary;

    public Employeee(String name, int id, double salary) {
        super();
        this.name = name;
        this.id = id;
        this.salary = salary;
    }

    @Override
    public String toString() {
        return "Employee{" + "name='" + name + '\'' + ", id=" + id + ", salary=" +
salary + '}';
    }

    public void increaseSalary(int percentage) {
        this.salary += salary * percentage / 100;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        ArrayList<Employeee> al = new ArrayList<>();
        al.add(new Employeee("Saurabh", 1, 50000));
        al.add(new Employeee("vikas", 2, 49000));
        al.add(new Employeee("Mayur", 3, 44000));
        al.add(new Employeee("Omkar", 4, 45900));
        System.out.println("Employees List : ");
        for (Employeee e : al) {
            System.out.println(e);
        }

        System.out.println("- - - - -");

        System.out.print("Enter ID to remove : ");

        int removeId = sc.nextInt();
        Iterator i = al.iterator();
        while (i.hasNext()) {
            Employeee e = (Employeee) i.next();
            if (e.id == removeId) {
                i.remove();
                System.out.println("Employee with ID : " + removeId + " has
Removed");
                break;
            }
        }
    }
}
```

```

        System.out.println("- - - - -");
    });
    System.out.println("Updated Employees List : ");
    for (Employee e : al) {
        System.out.println(e);
    }
}
}
}

```

```

Employees List :
Employee{name='Saurabh', id=1, salary=50000.0}
Employee{name='vikas', id=2, salary=49000.0}
Employee{name='Mayur', id=3, salary=44000.0}
Employee{name='Omkar', id=4, salary=45900.0}
- - - - -
Enter ID to remove : 2
Employee with ID :2 has Removed
- - - - -
Updated Employees List :
Employee{name='Saurabh', id=1, salary=50000.0}
Employee{name='Mayur', id=3, salary=44000.0}
Employee{name='Omkar', id=4, salary=45900.0}

```

Sort Method:

Sort method will sort the Object only when it is java.lang.Comparable type.

➤ Comparable Interface

java.lang.Comparable:

- It is an Interface that is defined in java.lang package.
- It is the functional Interface.

Abstract Methods of Comparable:

➤ Public abstract int compareTo(Object o)

To sort any object

- If the current object state is greater then passed object state should return positive integer
- If the current object state is less than passed object state it should return negative integer
- If the current object state is same to the passed object state it should return zero

Sorting the products based on the price:

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;

public class Products implements Comparable {
    String name;
    String brand;
    int pid;
    double price;

    public Products(String name, String brand, int pid, double price) {
        super();
        this.name = name;
        this.brand = brand;
        this.pid = pid;
        this.price = price;
    }

    public String toString() {
        return name;
    }

    public int compareTo(Object o) {
        Products p = (Products) o;
        if (pid > p.pid) {
            return 96;
        } else if (pid < p.pid) {
            return -89;
        } else {
            return 0;
        }
    }

    public static void main(String[] args) {
        ArrayList<Products> al = new ArrayList();
        al.add(new Products("Mobile", "Samsung", 83989, 70000.0));
        al.add(new Products("Light", "Philips", 5968, 1200));
        al.add(new Products("Shoe", "Bata", 839, 1500));
        al.add(new Products("Laptop", "HP", 89, 70000.0));

        Collections.sort(al);
        System.out.println(al);
    }
}
```

[Laptop, Shoe, Light, Mobile]

Sort the students based on names

```
package pack.subpack;

import java.lang.Comparable;
import java.util.ArrayList;
import java.util.Collections;

public class Students implements Comparable {
    String name;
    int rollNo;
    int rank;

    public Students(String name, int rollNo, int rank) {
        super();
        this.name = name;
        this.rollNo = rollNo;
        this.rank = rank;
    }

    public String toString() {
        return name;
    }

    public int compareTo(Object o) {
        Students s = (Students) o;
        if (rank > s.rank)
            return 89;
        else if (rank < s.rank)
            return -89;
        else
            return 0;
    }

    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(new Students("Saurabh", 226, 1));
        al.add(new Students("Govind", 275, 3));
        al.add(new Students("Mayur", 265, 5));
        al.add(new Students("Tanmay", 245, 4));
        al.add(new Students("Pratik", 285, 2));

        Collections.sort(al);
        System.out.println(al);
    }
}
```

```
<terminated> Students [java.Application; C:\Program Files\Java\jdk-20\bin\javaw.exe -
[Saurabh, Pratik, Govind, Tanmay, Mayur]
```

Sort the Book based on title

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;

public class Book implements Comparable {
    String title;
    String author;
    int pages;

    public Book(String title, String author, int pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }

    @Override
    public String toString() {
        return title;
    }

    @Override
    public int compareTo(Object o) {
        Book b = (Book) o;

        return title.compareTo(b.title);
    }

    public static void main(String[] args) {
        ArrayList<Book> al = new ArrayList<>();

        al.add(new Book("The Animal love", "Darshan", 561));
        al.add(new Book("Rules to become Romeo", "Saurabh", 541));
        al.add(new Book("Men will be Women ", "Vedant", 511));
        al.add(new Book("Must read in your 18", "Shiv", 461));

        Collections.sort(al);
        System.out.println(al);
    }
}
```

<terminated> Book [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May 2, 2024, 8:38:36 AM – 8:38:40 AM) [pid: 12524]

[Men will be Women , Must read in your 18, Rules to become Romeo, The Animal love]

Sort the bags based on colour

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;

public class Bag implements Comparable<Bag> {
    String color;
    int compartments;

    public Bag(String color, int compartments) {
        super();
        this.color = color;
        this.compartments = compartments;
    }

    @Override
    public String toString() {
        return color;
    }

    @Override
    public int compareTo(Bag b) {
        return -color.compareTo(b.color);
    }

    public static void main(String[] args) {
        ArrayList<Bag> al = new ArrayList<>();
        al.add(new Bag("Red", 4));
        al.add(new Bag("Green", 5));
        al.add(new Bag("Blue", 3));
        al.add(new Bag("Pink", 6));
        al.add(new Bag("Black", 4));

        Collections.sort(al);

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

```
[Red, Pink, Green, Blue, Black]
```

➤ **Comparator Interface**

Limitations of comparable:

- By using comparable interface objects can be sorted by using any one of the states
- To sort the objects based on different states is not possible i.e. multiple sorting option cannot be provided by comparable interface.
- If the objects are not comparable then sort method will not sort the objects in the list
- To overcome all the limitations of comparable interface java.util.Comparator can be used

java.util.Comparator

- It is an interface provided in java.util package
- It provides mechanism to sort the objects
- It is the functional interface.

Abstract Methods

➤ **Public abstract int compare(Object o1, Object o2);**

Characteristics:

- Even though objects are not comparable type by using comparator objects can be sorted
- By using comparator objects can be sorted by using multiple ways

sort (List l , comparator c) - method

TreeSet(Comparator c) – constructor

TreeMap(comparator c) – constructor

Uses the comparator mechanism to sort the objects

Steps to sort the object by using comparator:

1. Create a class and implement comparator interface
2. Override compare method
3. Call the **collections.sort ()** and pass the list as well as object of comparator type class

Example:

```
package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public final class Products {
    String name;
    String brand;
    int pid;
    double price;

    public Products(String name, String brand, int pid, double price) {
        super();
        this.name = name;
        this.brand = brand;
        this.pid = pid;
        this.price = price;
    }

    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList<Products> al = new ArrayList<>();
        al.add(new Products("Mobile", "Samsung", 83989, 70000.0));
        al.add(new Products("Light", "Philips", 5968, 1200));
        al.add(new Products("Shoe", "Bata", 839, 1500));
        al.add(new Products("Laptop", "HP", 89, 70000.0));

        System.out.println("Sorting based on price in asc order");
        Collections.sort(al, new SortByPriceAsc());
        System.out.println(al);

        System.out.println("Sorting based on price in desc order");
        Collections.sort(al, new SortByPricedesc());
        System.out.println(al);

        System.out.println("Sorting based on brand in asc order");
        Collections.sort(al, new SortByBrandAsc());
        System.out.println(al);
    }
}

class SortByPricedesc implements Comparator<Products> {
    @Override
    public int compare(Products p1, Products p2) {
        if (p1.price > p2.price)
            return -1;
        else if (p1.price < p2.price)
            return 1;
        else
            return 0;
    }
}
```

```

    }
}

class SortByBrandAsc implements Comparator<Products> {
    @Override
    public int compare(Products p1, Products p2) {
        return p1.brand.compareTo(p2.brand);
    }
}

class SortByPriceAsc implements Comparator<Products> {
    @Override
    public int compare(Products p1, Products p2) {
        return (int) (p1.price - p2.price);
    }
}

```

```

Sorting based on price in asc order
[Light, Shoe, Mobile, Laptop]
Sorting based on price in desc order
[Mobile, Laptop, Shoe, Light]
Sorting based on brand in asc order
[Shoe, Laptop, Light, Mobile]

```

```

package pack.subpack;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public final class Products {
    String name;
    String brand;
    int pid;
    double price;

    public Products(String name, String brand, int pid, double price) {
        super();
        this.name = name;
        this.brand = brand;
        this.pid = pid;
        this.price = price;
    }

    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        ArrayList<Products> al = new ArrayList<>();
        al.add(new Products("Mobile", "Samsung", 83989, 70000.0));
        al.add(new Products("Light", "Philips", 5968, 1200));
        al.add(new Products("Shoe", "Bata", 839, 1500));
        al.add(new Products("Laptop", "HP", 89, 70000.0));

        // lambda function/ expression
        System.out.println("Sorting based on price in asc order");
        Collections.sort(al, (p1,p2)-> (int)(p1.price-p2.price));
        System.out.println(al);

        System.out.println("Sorting based on brand");
        Collections.sort(al, (p1,p2)-> (p1.brand.compareTo(p2.brand)));
        System.out.println(al);

        System.out.println("Sorting based on name");
        Collections.sort(al, (p1,p2)-> p1.name.compareTo(p2.name));
        System.out.println(al);
    }
}

```

```

<terminated> Products [Java Application] C:\Program Files\Java\jdk-20\
Sorting based on price in asc order
[Light, Shoe, Mobile, Laptop]
Sorting based on brand
[Shoe, Laptop, Light, Mobile]
Sorting based on name
[Laptop, Light, Mobile, Shoe]

```


Linked List

Advantage of array list

Array list is best suitable for random accessing

Dis-Advantage of array list

In array list inserting the objects in the middle and removing the object takes more time because if any element is inserted or removed then remaining elements will shift their places.

Linked List

- It is implementing the subclass of **java.util.List** Interface
 - It is used to store multiple objects together
 - Linked list is implemented by using **doubly linked list** data structure
 - In linked list objects are stored in the form of **nodes** and one node is linked with the previous and the next node
-
- **Node:**
 - Node is an object
 - Every node contains an address, pointers and values
 - In doubly linked list every node contains two pointers and one value
 - One pointer is used to store the address of previous node and another pointer is used to store the address of next node

Address of previous node	Data	Address of next node
--------------------------	------	----------------------

The linked list is introduced in 1.2 version of JDK.

Characteristics

NIDHI

- 1) Any number of nulls can be inserted
- 2) In a list insertion order of an object is maintained
- 3) Duplicate objects are allowed
- 4) Heterogeneous objects can be inserted
- 5) Objects can be accessed by using an Index

Note:

In linked list to provide index support internally index pointer is created in doubly linked list data structure

Constructors

➤ **LinkedList()** :

It creates an empty linked list object

➤ **LinkedList(Collection c)** :

It creates a new linked list object that is initialized with objects in the specified collection

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

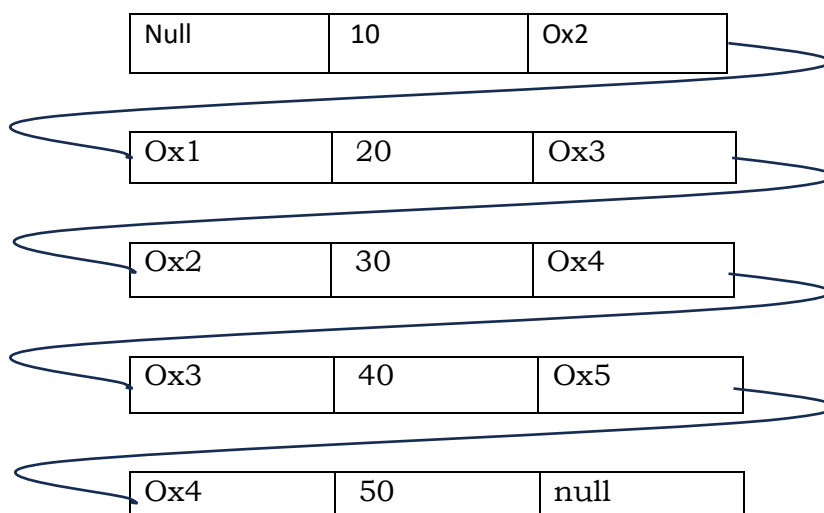
```
l.add ( 10 );
```

```
l.add ( 20 );
```

```
l.add ( 30 );
```

```
l.add ( 40 );
```

```
l.add ( 50 );
```



Crate food object and add in the linked list object

```
package pack.subpack;

import java.util.Collections;
import java.util.LinkedList;

public class Food {
    String name;
    int price;

    public Food(String name, int price) {
        super();
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        LinkedList<Food> l = new LinkedList<>();
        l.add(new Food("Vadapav", 12));
        l.add(new Food("Dabeli", 15));
        l.add(new Food("Panipuri", 20));
        l.add(new Food("Bhel", 18));
        l.add(new Food("Samosa", 22));

        System.out.println("Sorting based on the Name:");
        Collections.sort(l, (f1, f2) -> f1.name.compareTo(f2.name));
        System.out.println(l);

        System.out.println("Sorting based on the price:");
        Collections.sort(l, (f1, f2) -> f1.price - f2.price);
        System.out.println(l);
    }
}
```

```
<terminated> Food [Java Application] C:\Program Files\Java\jdk-20\bin\javaw.exe (May
Sorting based on the Name:
[Bhel, Dabeli, Panipuri, Samosa, Vadapav]
Sorting based on the price:
[Vadapav, Dabeli, Bhel, Panipuri, Samosa]
```

Vector

Java.util.Vector:

- It is an implementing class of java.util.List interface.
- It is introduced in 1.0 version hence, it is legacy class
- It is used to store multiple objects together
- It is implemented in global array data structure
- It is synchronized (one thread can access at a time)
- It also implements

Characteristics

NIDHI

- Multiple nulls can be inserted
- Heterogeneous objects are allowed
- Insertion order is maintained
- Duplicate objects are allowed
- Objects can be accessed by using index

Constructors:

- **Vector():** It creates an empty vector object with the default initial capacity of 10.
- **Vector(Collection c) :** It creates a new vector object that is initialized with the objects in the specified collection
- **Vector (int initialCapacity) :** It creates a new vector object with specified capacity.
- **Vector(int initialCapacity, int incrementCapacity):**
It creates an empty vector object with specified initial capacity and incremental capacity

Methods:

Vector contains all the methods of list interface and also its own legacy methods such as

- **Add(Object o)**
- **Capacity(),**
- **ElementAt(int index)**
- **firstElement()**
- **insertElementAt(Object o,int index)**

Note : It is recommended to use arraylist in place of vector

Difference between Vector and ArrayList

Feature	Vector	ArrayList
Synchronization	Vector is synchronized.	ArrayList is not synchronized by default.
Performance	Relatively slower due to synchronization.	Faster compared to Vector in non-threaded environment.
Growth	Doubles its size when it exceeds capacity.	Increases by 50% of its current size when it exceeds capacity.
Legacy	Part of the original Java collections framework.	Introduced in Java 2 (JDK 1.2) as part of the Collections Framework.
Thread Safety	Thread-safe.	Not thread-safe.
Iterator	Fail-safe iterator.	Fail-fast iterator.

Stack

Java.util.Stack

- It is the subclass of java.util.Vector
- Stack represents first in last out or last in first out of objects
- In stack elements are added from the top of stack and removed from the top of the stack
- It is introduced in **1.0** version of **JDK**

Constructor

- **Stack() :** **It creates an empty stack**

Methods

Stack contains the methods that are inherited from java.util class and its own methods.

- **Public Object push(Object o) :**

It is used to push the object into the top of stack

- **Public Object pop() :**

It is used to remove the object at the top of the stack and it return that object as a value

- **Public Object peek() :**

It is used to return the object at the top of the stack without removing

- **Public int search (Object o) :**

- It returns one based position from the stack .
- If the object is not present in the stack it return **-1**

- **Public boolean empty() :**

- It is used to return to check whether the stack is empty or not empty
- If the stack is empty it return true if no it return false.

```
package pack.subpack;

import java.util.Stack;

public class Stack1 {
    public static void main(String[] args) {
        Stack<Integer> s = new Stack();
        s.push(10);
        s.push(20);
        s.push(30);
        s.push(40);
        s.push(50);

        System.out.println(s);

        s.pop();
        s.pop();

        System.out.println(s);

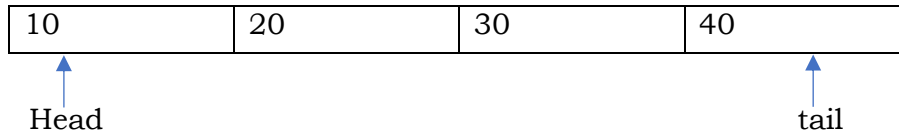
        System.out.println(s.peek());
        System.out.println(s.empty());
        System.out.println(s.search(10));
    }
}
```

```
terminated> Stack1.java Application: C:\p...
[10, 20, 30, 40, 50]
[10, 20, 30]
30
false
3
```

Queue

Java.util.Queue;

- It is an interface defined in java.util package
- It is the sub interface of java.util.Collection
- Queue follows **first in first out** or **last in last out(LILO)**
- In queue elements are added in tails side and removed or accessed from head side



- Implementing classes are:
 - PriorityQueue
 - Linkedlist
 - ArrayDeque
- It is introduced in 1.5 version of **JDK**

Abstract methods of queue:

1. Adding:

➤ **Public abstract Boolean add(Object o):**

- It is used to add an object into the queue from the tails side
- If the element is added it return true else it returns false and, in some cases, it throws exception

Cases:

- **ClassCastException:** If the specified object of the class is prevented from the queue it throws ClassCastException
- **NullPointerException:** if the specifies queue is not accepting null it throws NullPointerException

➤ **Public abstract boolean offer(Object o):**

- It is used to add the object into the queue from the tail side
- If the element is added it returns true else it returns false

2. Remove:

➤ Public abstract object remove():

- It is used to remove the object from the queue that is present in head
- Once after removing the object remove method returns the object
- If the queue is empty then the remove method throws NoSuchElementException.

➤ Public abstract object poll():

- It is used to remove the object the queue that is present in head
- Once after removing the object poll method returns the object
- If the queue is empty then the poll method returns null

3. Access:

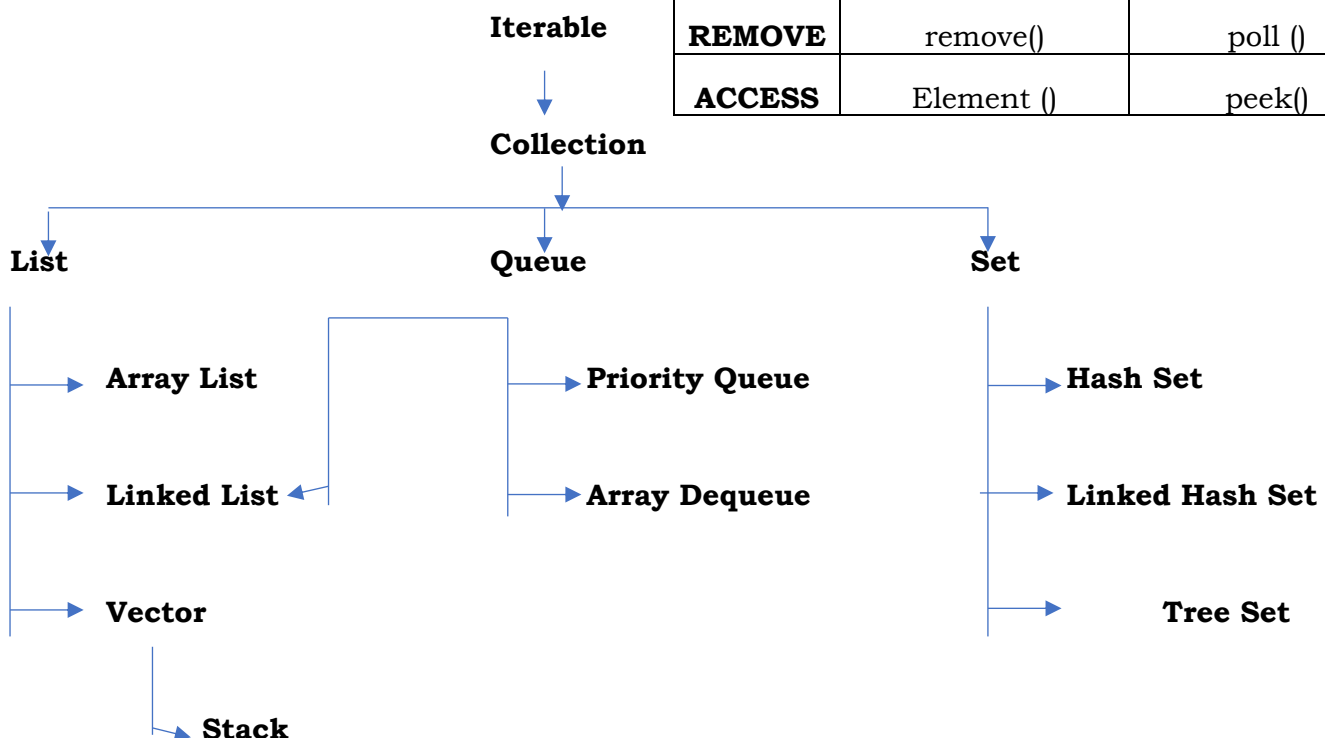
➤ Public abstract object element():

- It is used to return the object at the head of the queue.
- If the queue is empty then it throws NoSuchElementException.

➤ Public abstract object peek():

- It is used to return the object at the head of the queue.
- If the queue is empty then it returns null.

	Throws Exception	Value
ADD	add(Object o)	Offer (Object o)
REMOVE	remove()	poll ()
ACCESS	Element ()	peek()



Linked List

Java.util.LinkedList

- It is an implementing class of java.util.List and java.util.Queue interface
- In linked list elements are stored in the form of nodes where each node is linked with previous and next node
- Doubly linked list is the underlying data structure of linked list
- Linked list behaves like a list as well as the queue

```
package pack.subpack;

import java.util.Queue;
import java.util.LinkedList;
public class Queue1 {
    public static void main(String[] args) {
        LinkedList<Integer> l = new LinkedList<>();
        l.offer(10);
        l.offer(20);
        l.offer(30);
        l.offer(40);
        l.offer(50);

        System.out.println(l.peek());
        l.poll();
        l.poll();

        System.out.println(l.element());
        l.remove();
        System.out.println(l.element());
    }
}
```

```
10
30
40
```

ArrayDeque

Java.util.ArrayDeque:

- It implements java.util.Deque interface which is the child of queue interface
- ArrayDeque is implemented by using resizable array
- In array deque operations can be done in both end of the queue both head and tail end of the queue.
- Adding, removing and accessing can be done in both head and tail side of the queue.
- It is introduced in 1.6 version of JDK

Constructors:

➤ **ArrayDeque() :**

It creates an empty arrayDeque object with initial capacity of 16.

➤ **ArrayDeque(Collection c):**

It creates an arrayDeque object with the specified collection.

➤ **ArrayDeque(int initialCapacity):**

It creates an empty arrayDeque object with the specified initial capacity .

Characteristics:

- Null is not accepted
- Heterogeneous objects can be added
- Insertion order is maintained
- Duplicated objects are accepted
- Elements can not be accessed by using index.

Methods:

Functionality	Method Signature	Return Type
ADD	add(Object o)	boolean
	addFirst (Object o)	void
	addLast(Object o)	void
	Offer(Object o)	boolean
	OfferFirst(Object o)	boolean
	offerLast(Object o)	boolean
REMOVE	remove()	object
	removeFirst()	object
	removeLast()	object
	poll()	object
	pollFirst()	object
	pollLast()	object
ACCESS	element()	object
	getFirst()	object
	getLast()	object
	peek()	object
	peekFirst()	object
	peekLast()	object

```
package pack.subpack;

import java.util.ArrayDeque;
import java.util.Iterator;
public class Queue1 {
    public static void main(String[] args) {
        ArrayDeque<String> a = new ArrayDeque<>();

        a.offer("Shhela");
        a.offerFirst("Laila");
        a.addFirst("Mala");
        a.addLast("Sindhu");

        System.out.println(a.getLast());
        System.out.println(a.getFirst());
        System.out.println(a.peekFirst());
        a.pollLast();
        System.out.println(a);
    }
}
```

```
Sindhu
Mala
Mala
[Mala, Laila, Shhela]
```

Priority Queue

Java.util.PriorityQueue

- A priority queue in Java is an abstract data type.
- It functions similar to a regular queue or stack.
- However, it has a crucial distinction: each element has an associated priority.
- Elements with higher priorities are dequeued before those with lower priorities.
- Priority determines the order of dequeuing, not the order of insertion.
- Consequently, elements may be dequeued in a different order from their insertion order.

Key Points :

- It is implemented using a priority heap or binary heap.
- Elements are ordered based on their natural ordering (if they implement Comparable) or using a Comparator.
- The head of the priority queue is always the least (or highest, depending on ordering) element.
- Operations like insertion (enqueue) and removal of the head (dequeue) are performed efficiently in $O(\log n)$ time complexity.

Constructors:

1. Default Constructor:

Constructs an empty priority queue with an initial capacity of 11 elements. If more elements are added than this initial capacity, the priority queue will automatically resize itself.

```
PriorityQueue<E> pq = new PriorityQueue<>();
```

2. Constructor with Initial Capacity:

Constructs an empty priority queue with the specified initial capacity. The capacity is the number of elements the priority queue can initially store without resizing.

```
PriorityQueue<E> pq = new PriorityQueue<>(int initialCapacity);
```

3. Constructor with Comparator:

Constructs an empty priority queue with the specified initial capacity that orders its elements according to the specified comparator.

```
PriorityQueue<E> pq = new PriorityQueue<>(Comparator<? super E> comparator);
```

4. Constructor with Collection and Comparator:

Constructs a priority queue containing the elements of the specified collection, ordered according to the specified

```
PriorityQueue<E> pq = new PriorityQueue<>(Collection<? extends E> c, Comparator<? super E> comparator);
```

5. Constructor with Collection:

Constructs a priority queue containing the elements of the specified collection. The elements are ordered according to their natural ordering.

```
PriorityQueue<E> pq = new PriorityQueue<>(Collection<? extends E> c);
```

Here E represents the type of elements stored in the priority queue.

Abstract methods of priority queue:

➤ **Public abstract boolean add(object O):**

Insert the specified element into the priority queue.

➤ **Public abstract boolean offer(Object O) :**

Insert the specified element into the priority queue.

➤ **Public abstract boolean remove (object O):**

Removes a single instance of the specified element from this queue, if it is present.

➤ **Public abstract object poll ():**

Retrieves and removes the head of this queue, or returns null if this queue is empty.

➤ **Public abstract object peek():**

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

➤ **Public abstract object [] toArray():**

Returns an array containing all of the elements in this queue.

➤ **Public abstract boolean contains (object o):**

Returns true if this queue contains the specified element.

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        // Creating a PriorityQueue
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Adding elements to the PriorityQueue
        pq.add(10);
        pq.add(20);
        pq.add(15);

        // Printing the elements of the PriorityQueue
        System.out.println("Elements of PriorityQueue: " + pq);

        // Removing elements from the PriorityQueue
        int removedElement = pq.poll();
        System.out.println("Removed element: " + removedElement);

        // Printing the elements after removal
        System.out.println("Elements of PriorityQueue after removal: " + pq);

        // Peeking the head of the PriorityQueue
        int head = pq.peek();
        System.out.println("Head of PriorityQueue: " + head);
    }
}
```

Elements of PriorityQueue: [10, 20, 15]

Removed element: 10

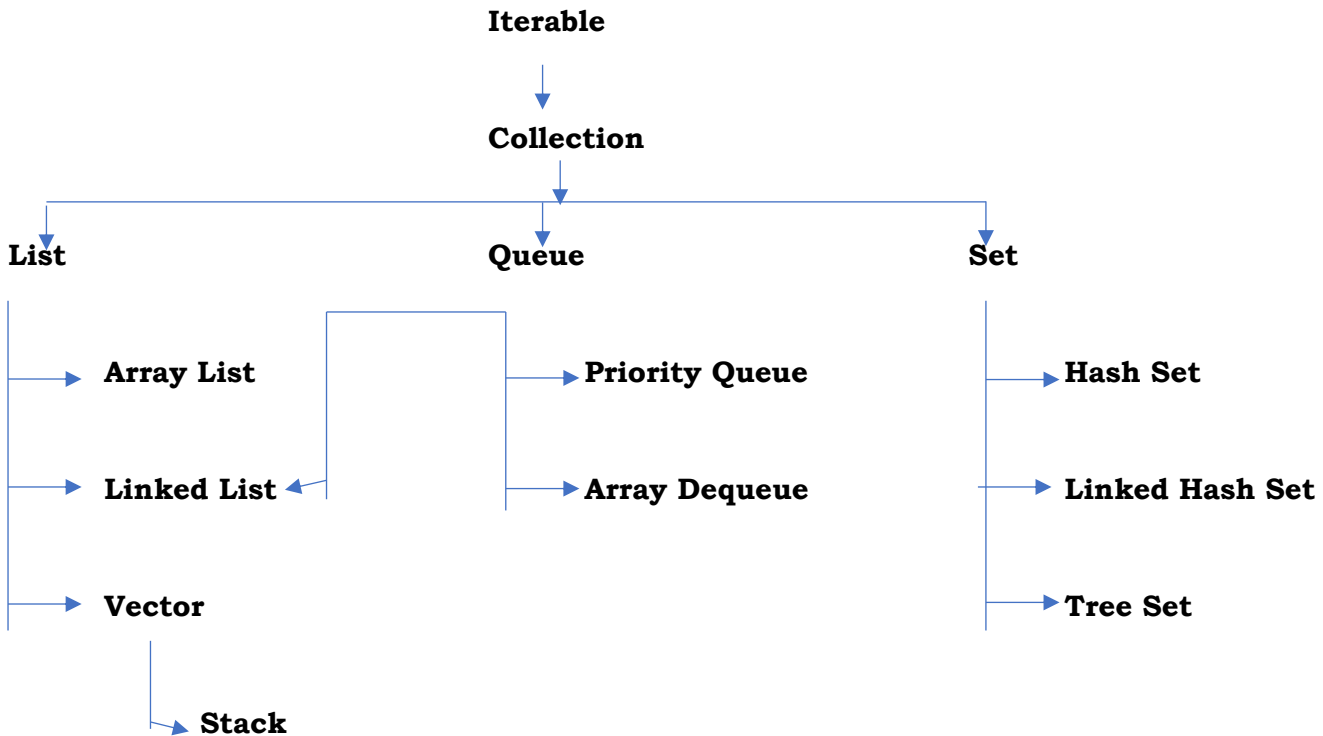
Elements of PriorityQueue after removal: [15, 20]

Head of PriorityQueue: 15

Set

Java.util.set

- It is an interface defined in java.util package
- It is the sub-interface of java.util.collection
- It is used to store multiple objects together
- Set doesn't allow duplicate objects and it is not index based
- HashSet, LinkedHashSet and TreeSet are the implementing classes of set interface



Java.util.HashSet

- It is the implementing class of java.util.Set interface.
- It is implementing by using hash Table
- It follows hashing mechanism to add the objects
- It is introduced in 1.2 version of **JDK**

Charateristics

NIDHI

- Only one null can be inserted
- Insertion order is not maintained
- Duplicate objects are not accepted
- Heterogeneous objects are allowed
- Elements cannot be accessed by using its index

Removing duplicate objects:

- HashSet removes the duplicate object with the help of hashCode and equals method.
- If the hashCode and equals method are not overridden then the duplicate objects are removed based on the addresses.
- If hashCode and equals method are overridden then the duplicate objects are removed based on the states of the objects

constructors

➤ **HashSet()**

It creates an empty hashset object with the default initial capacity of 16 and load factor of **0.75**

➤ **HashSet(int initialCapacity)**

It creates an empty HashSet object with specified initial capacity with load factor 0.75

➤ **HashSet(int initialCapacity , float loadFactor)**

It creates an empty HashSet object with specified initial capacity and load factor

➤ **HashSet(Collection c)**

It creates a HashSet Object with the specified collection

```
package pack.subpack;

import java.util.HashSet;
import java.util.Iterator;

public class HashSet1 {
    public static void main(String[] args) {
        HashSet hs = new HashSet<>();
        hs.add(20);
        hs.add("Sheela");
        hs.add(77.28);
        hs.add(99.22);
        hs.add(null);
        hs.add("Laila");
        hs.add("Laila");

        Iterator i = hs.iterator();

        while(i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Sheela
99.22
20
77.28
Laila

```
package pack.subpack;

import java.util.HashSet;
import java.util.Objects;

public class Students {
    String name;
    int id;

    public Students(String name, int id) {
        super();
        this.name = name;
        this.id = id;
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, id);
    }

    @Override
    public boolean equals(Object o) {
        Students s = (Students) o;
        if (name.equals(s.name) && id == s.id)
            return true;
        else
            return false;
    }

    public static void main(String[] args) {
        HashSet<Students> hs = new HashSet();
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

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

[Sindhu, Laila, Sheela, Shakela, Indhu]

Linked Hash set

Java.util.LinkedHashSet

- It is the subclass of java.util.HashSet.
- It is implemented by using hash table.
- In LinkedhashSet insertion order is maintained
- It is introduced in 1.4 version of JDK

Characteristics:

- Only one null can be inserted
- Insertion order is maintained
- Duplicate objects are not accepted
- Heterogeneous objects are allowed
- Elements cannot be accessed by using its index

Constructor

➤ **LinkedHashSet()**

It creates an empty LinkedHashSet object with the default initial capacity of 16 and load factor of **0.75**

➤ **LinkedHashSet(int initialCapacity)**

It creates an empty LinkedHashSet object with specified initial capacity with load factor 0.75

➤ **LinkedHashSet(int initialCapacity , float loadFactor)**

It creates an empty LinkedHashSet object with specified initial capacity and load factor

➤ **LinkedHashSet(Collection c)**

It creates a LinkedHashSet Object with the specified collection

Removing Duplicate objects

LinkedHashSet removes duplicate objects with the help of hashCode and equals method

```
package pack.subpack;

import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;

public class LinkedHashSet1 {
    public static void main(String[] args) {
        LinkedHashSet hs = new LinkedHashSet<>();
        hs.add(20);
        hs.add("Sheela");
        hs.add(77.28);
        hs.add(99.22);
        hs.add(null);
        hs.add("Laila");
        hs.add("Laila");

        Iterator i = hs.iterator();

        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

```
20
Sheela
77.28
99.22
null
Laila
```

```

package pack.subpack;

import java.util.LinkedHashSet;
import java.util.Objects;

public class Students {
    String name;
    int id;

    public Students(String name, int id) {
        super();
        this.name = name;
        this.id = id;
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, id);
    }

    @Override
    public boolean equals(Object o) {
        Students s = (Students) o;
        if (name.equals(s.name) && id == s.id)
            return true;
        else
            return false;
    }

    public static void main(String[] args) {
        LinkedHashSet<Students> hs = new LinkedHashSet();
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

        System.out.println(hs);
    }
}

```

```
[Sheela, Laila, Shakela, Sindhu, Indhu]
```

TreeSet

It is implementing class of java.util.set interface

In tree set by default all the objects are sorted based in natural sorting order

Hence, the object that are added into the tree set should be java.lang.comparable.

It is introduced in 1.2 version of **JDK**.

It is implemented by using navigable set and TreeMap

Characteristics:

- Null is not accepted
- If we try to add the null we will get null pointer Exception.
- Insertion order is not maintained
- Duplicate objects are not allowed
- Heterogeneous objects are nor accepted, if we try to add heterogeneous objects we will get class cast exception.
- Elements cannot be accessed by using its index

Note:

- **If the objects that are added to the tree set is not comparable type we get ClassCast Exception**
- **To add the object to the tree set which is not comparable we can use comparator**

Constructors:

➤ **TreeSet():**

It creates an empty treeset objects that is comparable and and it follows the natural ordering of an element for sorting

➤ **TreeSet(Collection c):**

It creates a new treeset objects that contains the objects in the specified collection which is sorted based on natural ordering that is comparable.

➤ **TreeSet(Comparator):**

It Creates an empty TreeSet object that sort all the objects based on specified comparator.

Note: when we use this constructor to add the object all the objects are sorted based on the comparator not based on the comparable

Example:

```
package pack.subpack;

import java.util.TreeSet;

public class Students implements Comparable<Students> {
    String name;
    int id;

    public Students(String name, int id) {
        super();
        this.name = name;
        this.id = id;
    }

    @Override
    public String toString() {
        return name;
    }

    @Override
    public int compareTo(Students s) {
        return id - s.id;
    }

    public static void main(String[] args) {
        TreeSet hs = new TreeSet();
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

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

```
[Sheela, Indhu, Laila, Shakela, Sindhu]
```


Sort By id in descending order:

```
package pack.subpack.set;

import java.util.Comparator;

public class SortByIdDesc implements Comparator<Students> {

    public int compare(Students o1, Students o2) {

        return -(o1.id - o2.id);

    }

}
```

```
package pack.subpack.set;

import java.util.TreeSet;

public class Students {
    String name;
    int id;

    public Students(String name, int id) {
        super();
        this.name = name;
        this.id = id;
    }

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        TreeSet hs = new TreeSet(new SortByIdDesc());
        hs.add(new Students("Sheela", 7));
        hs.add(new Students("Laila", 17));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Shakela", 18));
        hs.add(new Students("Sindhu", 71));
        hs.add(new Students("Indhu", 12));

        System.out.println(hs);

    }

}
```

[Sindhu, Shakela, Laila, Indhu, Sheela]

MAP

Java.util.Map

- It is an interface defined in java.util package
- It is used to store multiple objects together along with that for every object one unique identification is provided known as key
- Every key is associated with the values

E.g.:

District	–	pin code
Country	–	Country code
Employee	-	ID
Student	–	RollNo
Account	–	Ac/No.

- In map objects are stored in the form of key and value pairs.
- Key should be unique but value can be repeated
- One key and Value pair is known as entry

Key	value
-----	-------

Entry

- Hence, in a map multiple entries can be added

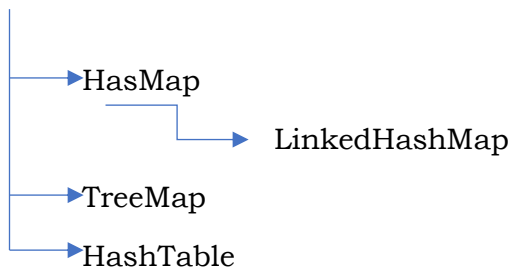
HashMap

LinkedHashMap

TreeMap

HashTable

Map



Abstract methods

These are the implementing classes of Map interface

Functionality	Method Name	Return Type
Adding Entries	put(Object key, Object Value)	Object (value)
	putAll(Map m)	void
Removing Entries	remove (Object Key)	Object (value)
	clear()	void
Search	containsKey(Object Key)	boolean
	containsValue (Object Value)	boolean
Replace	replace(Object Key, Object Value)	Object (value)
	replace(Object Key, Object OldValue, Object NewValue)	Object (value)
Access	get(Object Key)	Object (value)
	values()	Collection
	keySet()	Set
	entrySet()	Set

Java.util.HashMap

- It is implementing class of **Java.util.Map interface**.
- It is implemented by using HashTable
- It is introduced in **1.2 version of JDK**
- It is used to store the objects in the form of key and value pairs

Characteristics:

- Only one null can be used as a key
- Insertion order of entry is not maintained
- Duplicate keys not allowed , If we try to use duplicate keys then the existing values is replaced with new value
- Heterogeneous keys can be inserted
- It is not index based

Constructors:

➤ **HashMap():**

Initial capacity is 16.

Load factor is 0.75.

➤ **HashMap(int initialCapacity):**

Default load factor is 0.75

➤ **HashMap(int initialCapacity, float loadfactor):**

➤ **HashMap(Map m):**

Example:

```
package pack.subpack.set;

import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap hs = new HashMap();

        hs.put(5, "Sheela");
        hs.put(6, "Laila");
        hs.put(true, "Job");
        hs.put(7.5, 78);
        hs.put('i', 79);
        hs.put("One", 1);
        hs.put(null, "nothing");
        System.out.println(hs);
        hs.put(5, "Sindhu");//It replaces Sheela with Sindhu

        System.out.println(hs);

        System.out.println(hs.values());
        System.out.println(hs.keySet());
        System.out.println(hs.entrySet());
    }
}
```

```
{null=nothing, 5=Sheela, 6=Laila, One=1, i=79, 7.5=78, true=Job}
{null=nothing, 5=Sindhu, 6=Laila, One=1, i=79, 7.5=78, true=Job}
[nothing, Sindhu, Laila, 1, 79, 78, Job]
[null, 5, 6, One, i, 7.5, true]
[null=nothing, 5=Sindhu, 6=Laila, One=1, i=79, 7.5=78, true=Job]
```