Topics to cover:

1. Classes in collection framework

1. Classes in collection framework

In our previous tutorial we saw that there are many classes that implements and/or extends other interfaces and/or classes. We have already covered some of them in our previous classes such as ArrayList and HashMap. In this tutorial we will discuss the other most useful classes in the collection framework such as follows:

1.15 EnumMap class

1.16 Collections class

1.17 Comparable interface

1.18 Comparator interface

1.19 Properties class

1.20 Vector class

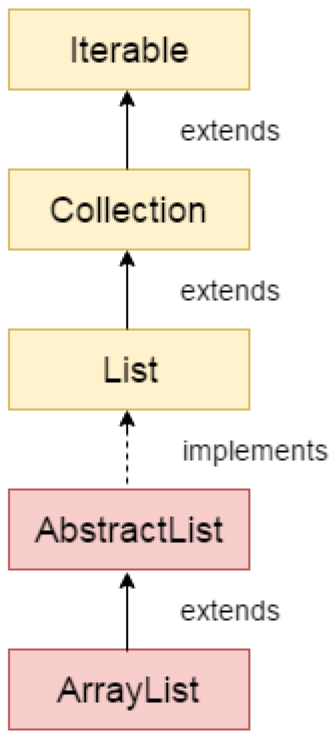
In this course we will discuss some of the above classes and interfaces. For more details, click on the following link:

https://www.javatpoint.com/collections-in-java

1.1 ArrayList

Java **ArrayList** class uses a *dynamic array* for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.



It inherits the AbstractList class and implements List interface.

The important points about the Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

```
ArrayList<int> al = ArrayList<int>(); // does not work
ArrayList<Integer> al = new ArrayList<Integer>(); // works fine
```

- Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

Hierarchy of ArrayList class

As shown in the above diagram, the Java ArrayList class extends AbstractList class which implements the List interface. The List interface extends the Collection and Iterable interfaces in hierarchical order.

Java Non-generic Vs. Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.

Let's see the old non-generic example of creating a Java collection.

```
ArrayList list=new ArrayList();//creating old non-generic arraylist
```

Let's see the new generic example of creating java collection.

```
ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist
```

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of object in it. If you try to add another type of object, it gives a *compile-time error*.

For more information on Java generics, click here Java Generics Tutorial.

Java ArrayList Example

FileName: ArrayListExample1.java

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
      list.add("Mango");//Adding object in arraylist
      list.add("Apple");
      list.add("Banana");
      list.add("Grapes");
      //Printing the arraylist object
      System.out.println(list);
 }
}
```

Output:

```
[Mango, Apple, Banana, Grapes]
```

Iterating ArrayList using Iterator

Let's see an example to traverse ArrayList elements using the Iterator interface.

FileName: ArrayListExample2.java

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and move to next
  }
 }
}
```

Output:

```
Mango
Apple
Banana
Grapes
```

Iterating ArrayList using For-each loop

Let's see an example to traverse the ArrayList elements using the for-each loop

FileName: ArrayListExample3.java

```
import java.util.*;
public class ArrayListExample3{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through for-each loop
  for(String fruit:list)
    System.out.println(fruit);

 }
}
```

Output:

```
Mango
Apple
Banana
Grapes
```

Get and Set ArrayList

The *get() method* returns the element at the specified index, whereas the *set() method* changes the element.

FileName: ArrayListExample4.java

```
import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Mango");
  al.add("Apple");
  al.add("Banana");
  al.add("Grapes");
  //accessing the element
    System.out.println("Returning element: "+al.get(1));//it will return the 2nd element,
because index starts from 0
  //changing the element
  al.set(1,"Dates");
  //Traversing list
  for(String fruit:al)
    System.out.println(fruit);

 }
}
```

Output:

```
Returning element: Apple
Mango
Dates
Banana
Grapes
```

How to Sort ArrayList

The *java.util* package provides a utility class **Collections**, which has the static method sort(). Using the **Collections.sort()** method, we can easily sort the ArrayList.

FileName: SortArrayList.java

```
import java.util.*;
class SortArrayList{
 public static void main(String args[]){
  //Creating a list of fruits
  List<String> list1=new ArrayList<String>();
  list1.add("Mango");
  list1.add("Apple");
  list1.add("Banana");
  list1.add("Grapes");
  //Sorting the list
  Collections.sort(list1);
   //Traversing list through the for-each loop
  for(String fruit:list1)
    System.out.println(fruit);

 System.out.println("Sorting numbers...");
  //Creating a list of numbers
  List<Integer> list2=new ArrayList<Integer>();
  list2.add(21);
  list2.add(11);
  list2.add(51);
  list2.add(1);
  //Sorting the list
  Collections.sort(list2);
   //Traversing list through the for-each loop
  for(Integer number:list2)
    System.out.println(number);
 }

}
```

```
Output:

Apple
Banana
Grapes
Mango
Sorting numbers...
1
11
21
51
```

Ways to iterate the elements of the collection in Java

There are various ways to traverse the collection elements:

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.

Iterating Collection through remaining ways

Let's see an example to traverse the ArrayList elements through other ways

FileName: ArrayList4.java

```
import java.util.*;
class ArrayList4{
 public static void main(String args[]){
    ArrayList<String> list=new ArrayList<String>();//Creating arraylist
           list.add("Ravi");//Adding object in arraylist
           list.add("Vijay");
           list.add("Ravi");
           list.add("Ajay");

           System.out.println("Traversing list through List Iterator:");
           //Here, element iterates in reverse order
              ListIterator<String> list1=list.listIterator(list.size());
              while(list1.hasPrevious())
              {
                  String str=list1.previous();
                  System.out.println(str);
              }
        System.out.println("Traversing list through for loop:");
           for(int i=0;i<list.size();i++)
           {
            System.out.println(list.get(i));
           }

        System.out.println("Traversing list through forEach() method:");
        //The forEach() method is a new feature, introduced in Java 8.
            list.forEach(a->{ //Here, we are using lambda expression
                System.out.println(a);
              });

            System.out.println("Traversing list through forEachRemaining() method:");
              Iterator<String> itr=list.iterator();
              itr.forEachRemaining(a-> //Here, we are using lambda expression
              {
```

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

```
Output:

Traversing list through List Iterator:
Ajay
Ravi
Vijay
Ravi
Traversing list through for loop:
Ravi
Vijay
Ravi
Ajay
Traversing list through forEach() method:
Ravi
Vijay
Ravi
Ajay
Traversing list through forEachRemaining() method:
Ravi
Vijay
Ravi
Ajay
```

User-defined class objects in Java ArrayList

Let's see an example where we are storing Student class object in an array list.

FileName: ArrayList5.java

```
class Student{
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age){
   this.rollno=rollno;
   this.name=name;
   this.age=age;
  }
}

import java.util.*;
 class ArrayList5{
 public static void main(String args[]){
  //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);
  //Getting Iterator
  Iterator itr=al.iterator();
  //traversing elements of ArrayList object
  while(itr.hasNext()){
    Student st=(Student)itr.next();
    System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 }
}
```

Output:

```
101 Sonoo 23
102 Ravi 21
103 Hanumat 25
```

Java ArrayList example to add elements

Here, we see different ways to add an element.

FileName: ArrayList7.java

```
import java.util.*;
 class ArrayList7{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
          System.out.println("Initial list of elements: "+al);
          //Adding elements to the end of the list
          al.add("Ravi");
          al.add("Vijay");
          al.add("Ajay");
          System.out.println("After invoking add(E e) method: "+al);
          //Adding an element at the specific position
          al.add(1, "Gaurav");
          System.out.println("After invoking add(int index, E element) method: "+al);
          ArrayList<String> al2=new ArrayList<String>();
          al2.add("Sonoo");
          al2.add("Hanumat");
          //Adding second list elements to the first list
          al.addAll(al2);
                System.out.println("After invoking addAll(Collection<? extends E> c) method:
"+al);
          ArrayList<String> al3=new ArrayList<String>();
          al3.add("John");
          al3.add("Rahul");
          //Adding second list elements to the first list at specific position
          al.addAll(1, al3);
               System.out.println("After invoking addAll(int index, Collection<? extends E> c)
method: "+al);

 }
}
```

```
Output:

Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After invoking addAll(Collection<? extends E> c) method:
[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addAll(int index, Collection<? extends E> c) method:
[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
```

Java ArrayList example to remove elements

Here, we see different ways to remove an element.

FileName: ArrayList8.java

```
import java.util.*;
 class ArrayList8 {

        public static void main(String [] args)
        {
          ArrayList<String> al=new ArrayList<String>();
          al.add("Ravi");
          al.add("Vijay");
          al.add("Ajay");
          al.add("Anuj");
          al.add("Gaurav");
          System.out.println("An initial list of elements: "+al);
          //Removing specific element from arraylist
          al.remove("Vijay");
          System.out.println("After invoking remove(object) method: "+al);
          //Removing element on the basis of specific position
          al.remove(0);
          System.out.println("After invoking remove(index) method: "+al);
          //Creating another arraylist
          ArrayList<String> al2=new ArrayList<String>();
          al2.add("Ravi");
          al2.add("Hanumat");
          //Adding new elements to arraylist
          al.addAll(al2);
          System.out.println("Updated list : "+al);
          //Removing all the new elements from arraylist
          al.removeAll(al2);
          System.out.println("After invoking removeAll() method: "+al);
          //Removing elements on the basis of specified condition
          al.removeIf(str -> str.contains("Ajay"));   //Here, we are using Lambda expression
          System.out.println("After invoking removeIf() method: "+al);
          //Removing all the elements available in the list
          al.clear();
          System.out.println("After invoking clear() method: "+al);
        }
     }
```

Output:

```
An initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav]
After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav]
After invoking remove(index) method: [Ajay, Anuj, Gaurav]
Updated list : [Ajay, Anuj, Gaurav, Ravi, Hanumat]
After invoking removeAll() method: [Ajay, Anuj, Gaurav]
After invoking removeIf() method: [Anuj, Gaurav]
After invoking clear() method: []
```

Java ArrayList example of retainAll() method

FileName: ArrayList9.java

```
import java.util.*;
class ArrayList9{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
  ArrayList<String> al2=new ArrayList<String>();
  al2.add("Ravi");
  al2.add("Hanumat");
  al.retainAll(al2);
  System.out.println("iterating the elements after retaining the elements of al2");
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
       iterating the elements after retaining the elements of al2
       Ravi
```

Java ArrayList example of isEmpty() method

FileName: ArrayList4.java

```
import java.util.*;
 class ArrayList10{

        public static void main(String [] args)
        {
          ArrayList<String> al=new ArrayList<String>();
          System.out.println("Is ArrayList Empty: "+al.isEmpty());
          al.add("Ravi");
          al.add("Vijay");
          al.add("Ajay");
          System.out.println("After Insertion");
          System.out.println("Is ArrayList Empty: "+al.isEmpty());
        }
     }
```

Output:

```
Is ArrayList Empty: true
After Insertion
Is ArrayList Empty: false
```

Java ArrayList Example: Book

Let's see an ArrayList example where we are adding books to the list and printing all the books.

FileName: ArrayListExample20.java

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
}
public class ArrayListExample20 {
public static void main(String[] args) {
   //Creating list of Books
   List<Book> list=new ArrayList<Book>();
   //Creating Books
   Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
```

```
     Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw
Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to list
    list.add(b1);
    list.add(b2);
    list.add(b3);
    //Traversing list
    for(Book b:list){
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications and Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

Size and Capacity of an ArrayList

Size and capacity of an array list are the two terms that beginners find confusing. Let's understand it in this section with the help of some examples. Consider the following code snippet.

FileName: SizeCapacity.java

```
import java.util.*;

public class SizeCapacity
{

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

    System.out.println("The size of the array is: " + al.size());
}
}
```

Output:

```
The size of the array is: 0
```

Explanation: The output makes sense as we have not done anything with the array list. Now observe the following program.

FileName: SizeCapacity1.java

```
import java.util.*;

public class SizeCapacity1
{

public static void main(String[] args) throws Exception
{

    ArrayList<Integer> al = new ArrayList<Integer>(10);

    System.out.println("The size of the array is: " + al.size());
}
}
```

Output:

```
The size of the array is: 0
```

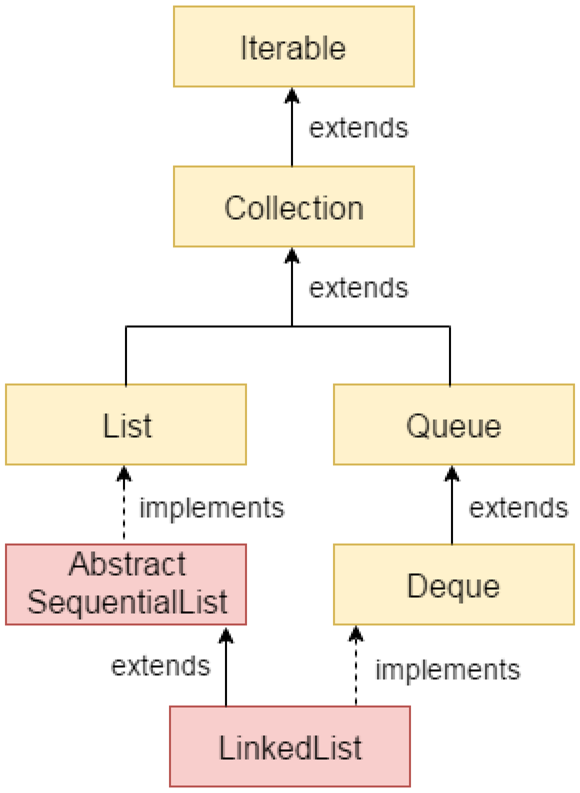
Explanation: We see that the size is still 0, and the reason behind this is the number 10 represents the capacity no the size. In fact, the size represents the total number of elements present in the array. As we have not added any element, therefore, the size of the array list is zero in both programs.

Capacity represents the total number of elements the array list can contain. Therefore, the capacity of an array list is always greater than or equal to the size of the array list. When we add an element to the array list, it checks whether the size of the array list has become equal to the capacity or not. If yes, then the capacity of the array list increases. So, in the above example, the capacity will be 10 till 10 elements are added to the list. When we add the 11th element, the capacity increases. Note that in both examples, the capacity of the array list is 10. In the first case, the capacity is 10 because the default capacity of the array list is 10. In the second case, we have explicitly mentioned that the capacity of the array list is 10.

Note: There is no any standard method to tell how the capacity increases in the array list. In fact, the way the capacity increases vary from one GDK version to the other version. Therefore, it is required to check the way capacity increases code is implemented in the GDK. There is no any pre-defined method in the ArrayList class that returns the capacity of the array list. Therefore, for better understanding, use the capacity() method of the Vector class. The logic of the size and the capacity is the same in the ArrayList class and the Vector class.

1.2 LinkedList class

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.



The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

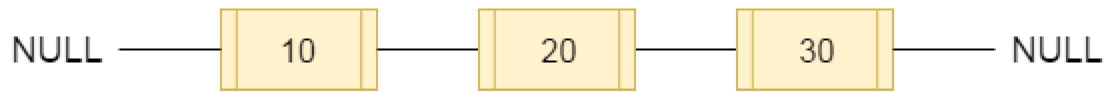


fig- doubly linked list

Java LinkedList Example

```
import java.util.*;
public class LinkedList1{
 public static void main(String args[]){

  LinkedList<String> al=new LinkedList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");

  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Output: Ravi
       Vijay
       Ravi
       Ajay
```

Java LinkedList example to add elements

Here, we see different ways to add elements.

```
import java.util.*;
public class LinkedList2{
 public static void main(String args[]){
 LinkedList<String> ll=new LinkedList<String>();
          System.out.println("Initial list of elements: "+ll);
          ll.add("Ravi");
          ll.add("Vijay");
          ll.add("Ajay");
          System.out.println("After invoking add(E e) method: "+ll);
          //Adding an element at the specific position
          ll.add(1, "Gaurav");
          System.out.println("After invoking add(int index, E element) method: "+ll);
          LinkedList<String> ll2=new LinkedList<String>();
          ll2.add("Sonoo");
          ll2.add("Hanumat");
          //Adding second list elements to the first list
          ll.addAll(ll2);
          System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
          LinkedList<String> ll3=new LinkedList<String>();
          ll3.add("John");
          ll3.add("Rahul");
          //Adding second list elements to the first list at specific position
          ll.addAll(1, ll3);
          System.out.println("After invoking addAll(int index, Collection<? extends E> c) method:
"+ll);
          //Adding an element at the first position
          ll.addFirst("Lokesh");
          System.out.println("After invoking addFirst(E e) method: "+ll);
          //Adding an element at the last position
          ll.addLast("Harsh");
          System.out.println("After invoking addLast(E e) method: "+ll);

 }
}
```

Output:

```
Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After invoking addAll(Collection<? extends E> c) method:
[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addAll(int index, Collection<? extends E> c) method:
[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addFirst(E e) method:
[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
```

```
After invoking addLast(E e) method:
[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]
```

Java LinkedList example to remove elements

Here, we see different ways to remove an element.

```
import java.util.*;
public class LinkedList3 {

        public static void main(String [] args)
        {
           LinkedList<String> ll=new LinkedList<String>();
           ll.add("Ravi");
           ll.add("Vijay");
           ll.add("Ajay");
           ll.add("Anuj");
           ll.add("Gaurav");
           ll.add("Harsh");
           ll.add("Virat");
           ll.add("Gaurav");
           ll.add("Harsh");
           ll.add("Amit");
           System.out.println("Initial list of elements: "+ll);
         //Removing specific element from arraylist
              ll.remove("Vijay");
              System.out.println("After invoking remove(object) method: "+ll);
         //Removing element on the basis of specific position
              ll.remove(0);
              System.out.println("After invoking remove(index) method: "+ll);
              LinkedList<String> ll2=new LinkedList<String>();
              ll2.add("Ravi");
              ll2.add("Hanumat");
         // Adding new elements to arraylist
              ll.addAll(ll2);
              System.out.println("Updated list : "+ll);
         //Removing all the new elements from arraylist
              ll.removeAll(ll2);
              System.out.println("After invoking removeAll() method: "+ll);
         //Removing first element from the list
```

```
            ll.removeFirst();
            System.out.println("After invoking removeFirst() method: "+ll);
        //Removing first element from the list
            ll.removeLast();
            System.out.println("After invoking removeLast() method: "+ll);
        //Removing first occurrence of element from the list
            ll.removeFirstOccurrence("Gaurav");
            System.out.println("After invoking removeFirstOccurrence() method: "+ll);
        //Removing last occurrence of element from the list
            ll.removeLastOccurrence("Harsh");
            System.out.println("After invoking removeLastOccurrence() method: "+ll);

            //Removing all the elements available in the list
            ll.clear();
            System.out.println("After invoking clear() method: "+ll);
      }
   }
```

Output:

```
Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh,
Amit]
After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]
After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]
After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]
After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]
After invoking clear() method: []
```

Java LinkedList Example to reverse a list of elements

```
import java.util.*;
public class LinkedList4{
 public static void main(String args[]){

  LinkedList<String> ll=new LinkedList<String>();
           ll.add("Ravi");
           ll.add("Vijay");
           ll.add("Ajay");
```

```
          //Traversing the list of elements in reverse order
          Iterator i=ll.descendingIterator();
          while(i.hasNext())
          {
             System.out.println(i.next());
          }

 }
}
```

```
Output: Ajay
Vijay
Ravi
```

Java LinkedList Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
}
public class LinkedListExample {
public static void main(String[] args) {
   //Creating list of Books
   List<Book> list=new LinkedList<Book>();
   //Creating Books
   Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
   Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
   Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
   //Adding Books to list
   list.add(b1);
   list.add(b2);
   list.add(b3);
```

```
    //Traversing list
    for(Book b:list){
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

Difference Between ArrayList and LinkedList

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

However, there are many differences between the ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.

3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

Example of ArrayList and LinkedList in Java

Let's see a simple example where we are using ArrayList and LinkedList both.

FileName: TestArrayLinked.java

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

  List<String> al=new ArrayList<String>();//creating arraylist
  al.add("Ravi");//adding object in arraylist
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");


  List<String> al2=new LinkedList<String>();//creating linkedlist
  al2.add("James");//adding object in linkedlist
  al2.add("Serena");
  al2.add("Swati");
  al2.add("Junaid");


  System.out.println("arraylist: "+al);
  System.out.println("linkedlist: "+al2);
 }
}
```

Output:

```
arraylist: [Ravi,Vijay,Ravi,Ajay]
linkedlist: [James,Serena,Swati,Junaid]
```

Points to Remember

The following are some important points to remember regarding an ArrayList and LinkedList.

- When the rate of addition or removal rate is more than the read scenarios, then go for the LinkedList. On the other hand, when the frequency of the read scenarios is more than the addition or removal rate, then ArrayList takes precedence over LinkedList.

- Since the elements of an ArrayList are stored more compact as compared to a LinkedList; therefore, the ArrayList is more cache-friendly as compared to the LinkedList. Thus, chances for the cache miss are less in an ArrayList as compared to a LinkedList. Generally, it is considered that a LinkedList is poor in cache-locality.
- Memory overhead in the LinkedList is more as compared to the ArrayList. It is because, in a LinkedList, we have two extra links (next and previous) as it is required to store the address of the previous and the next nodes, and these links consume extra space. Such links are not present in an ArrayList.

1.3 List interface

List in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

List Interface declaration

```
public interface List<E> extends Collection<E>
```

Java List vs ArrayList

List is an interface whereas ArrayList is the implementation class of List.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

```
//Creating a List of type String using ArrayList
List<String> list=new ArrayList<String>();

//Creating a List of type Integer using ArrayList
List<Integer> list=new ArrayList<Integer>();

//Creating a List of type Book using ArrayList
List<Book> list=new ArrayList<Book>();

//Creating a List of type String using LinkedList
List<String> list=new LinkedList<String>();
```

In short, you can create the List of any type. The ArrayList<T> and LinkedList<T> classes are used to specify the type. Here, T denotes the type.

Java List Example

Let's see a simple example of List where we are using the ArrayList class as the implementation.

```
import java.util.*;

public class ListExample1{

public static void main(String args[]){

 //Creating a List

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

 //Adding elements in the List

 list.add("Mango");

 list.add("Apple");

 list.add("Banana");
 list.add("Grapes");
 //Iterating the List element using for-each loop
 for(String fruit:list)
  System.out.println(fruit);

}
}
```

```
Output:
Mango
Apple
Banana
Grapes
```

How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

```
import java.util.*;
public class ArrayToListExample{
public static void main(String args[]){
//Creating Array
String[] array={"Java","Python","PHP","C++"};
System.out.println("Printing Array: "+Arrays.toString(array));

//Converting Array to List

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

for(String lang:array){

list.add(lang);

}

System.out.println("Printing List: "+list);


}

}
```

Output:

```
Printing Array: [Java, Python, PHP, C++]

Printing List: [Java, Python, PHP, C++]
```

How to convert List to Array

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

```
import java.util.*;

public class ListToArrayExample{

public static void main(String args[]){

 List<String> fruitList = new ArrayList<>();
 fruitList.add("Mango");
 fruitList.add("Banana");
 fruitList.add("Apple");
 fruitList.add("Strawberry");
 //Converting ArrayList to Array
 String[] array = fruitList.toArray(new String[fruitList.size()]);
 System.out.println("Printing Array: "+Arrays.toString(array));
 System.out.println("Printing List: "+fruitList);
}
}
```

Output:

```
Printing Array: [Mango, Banana, Apple, Strawberry]
Printing List: [Mango, Banana, Apple, Strawberry]
```

Get and Set Element in List

The *get() method* returns the element at the given index, whereas the *set() method* changes or replaces the element.

```
import java.util.*;
public class ListExample2{
 public static void main(String args[]){
```

```
 //Creating a List

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

 //Adding elements in the List

 list.add("Mango");

 list.add("Apple");

 list.add("Banana");

 list.add("Grapes");

 //accessing the element

 System.out.println("Returning element: "+list.get(1));//it will return the 2nd element, because
index starts from 0

 //changing the element

 list.set(1,"Dates");

 //Iterating the List element using for-each loop

 for(String fruit:list)

  System.out.println(fruit);


 }

}
```

Output:

```
Returning element: Apple

Mango
```

```
Dates

Banana

Grapes
```

How to Sort List

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The *java.util* package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort any List.

```
import java.util.*;

class SortArrayList{

 public static void main(String args[]){

  //Creating a list of fruits

  List<String> list1=new ArrayList<String>();

  list1.add("Mango");

  list1.add("Apple");

  list1.add("Banana");

  list1.add("Grapes");

  //Sorting the list

  Collections.sort(list1);

   //Traversing list through the for-each loop

  for(String fruit:list1)

    System.out.println(fruit);
 System.out.println("Sorting numbers...");
  //Creating a list of numbers
  List<Integer> list2=new ArrayList<Integer>();
  list2.add(21);
  list2.add(11);
  list2.add(51);
  list2.add(1);
  //Sorting the list
  Collections.sort(list2);
   //Traversing list through the for-each loop
  for(Integer number:list2)
    System.out.println(number);
 }

}
```

Output:

```
Apple
Banana
Grapes
Mango
Sorting numbers...
1
```

```
11

21

51
```

Java ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.

ListIterator Interface declaration

```
public interface ListIterator<E> extends Iterator<E>
```

Example of ListIterator Interface

```
import java.util.*;

public class ListIteratorExample1{

public static void main(String args[]){

List<String> al=new ArrayList<String>();

    al.add("Amit");

    al.add("Vijay");

    al.add("Kumar");

    al.add(1,"Sachin");

    ListIterator<String> itr=al.listIterator();

    System.out.println("Traversing elements in forward direction");

    while(itr.hasNext()){
    System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());
    }
    System.out.println("Traversing elements in backward direction");
    while(itr.hasPrevious()){

    System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());
    }
}
}
```

```
Output:
Traversing elements in forward direction
index:0 value:Amit
index:1 value:Sachin
index:2 value:Vijay
index:3 value:Kumar
Traversing elements in backward direction
index:3 value:Kumar
index:2 value:Vijay
index:1 value:Sachin
index:0 value:Amit
```

Example of List: Book

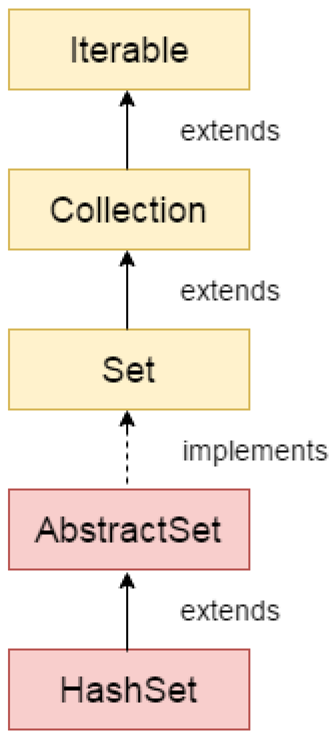
Let's see an example of List where we are adding the Books.

```
import java.util.*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;

}

}

public class ListExample5 {

public static void main(String[] args) {

    //Creating list of Books

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

    //Creating Books

    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

    Book b2=new Book(102,"Data Communications and Networking","Forouzan","Mc Graw
Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

    //Adding Books to list

    list.add(b1);

    list.add(b2);
    list.add(b3);

    //Traversing list

    for(Book b:list){

    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

    }

}

}
```

```
Output:


101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6
```

1.4 HashSet class

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.



The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

Java HashSet Example

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet();
           set.add("One");
           set.add("Two");
           set.add("Three");
           set.add("Four");
           set.add("Five");
           Iterator<String> i=set.iterator();
           while(i.hasNext())
           {
           System.out.println(i.next());
           }
 }
}
```

```
Output:

Five
One
Four
Two
Three
```

Java HashSet example ignoring duplicate elements

In this example, we see that HashSet doesn't allow duplicate elements.

```
import java.util.*;
class HashSet2{
 public static void main(String args[]){
  //Creating HashSet and adding elements
  HashSet<String> set=new HashSet<String>();
  set.add("Ravi");
  set.add("Vijay");
  set.add("Ravi");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
       Ajay
       Vijay
       Ravi
```

Java HashSet example to remove elements

Here, we see different ways to remove an element.

```
import java.util.*;
class HashSet3{
 public static void main(String args[]){
  HashSet<String> set=new HashSet<String>();
           set.add("Ravi");
           set.add("Vijay");
           set.add("Arun");
           set.add("Sumit");
           System.out.println("An initial list of elements: "+set);
           //Removing specific element from HashSet
           set.remove("Ravi");
           System.out.println("After invoking remove(object) method: "+set);
```

```
        HashSet<String> set1=new HashSet<String>();

        set1.add("Ajay");

        set1.add("Gaurav");

        set.addAll(set1);

        System.out.println("Updated List: "+set);

        //Removing all the new elements from HashSet

        set.removeAll(set1);

        System.out.println("After invoking removeAll() method: "+set);

        //Removing elements on the basis of specified condition

        set.removeIf(str->str.contains("Vijay"));

        System.out.println("After invoking removeIf() method: "+set);

        //Removing all the elements available in the set

        set.clear();

        System.out.println("After invoking clear() method: "+set);

 }

}
```

Output:

```
An initial list of elements: [Vijay, Ravi, Arun, Sumit]

After invoking remove(object) method: [Vijay, Arun, Sumit]

Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]
```

```
After invoking removeAll() method: [Vijay, Arun, Sumit]

After invoking removeIf() method: [Arun, Sumit]

After invoking clear() method: []
```

Java HashSet from another Collection

```
import java.util.*;

class HashSet4{

 public static void main(String args[]){

   ArrayList<String> list=new ArrayList<String>();

           list.add("Ravi");

           list.add("Vijay");

           list.add("Ajay");


           HashSet<String> set=new HashSet(list);

           set.add("Gaurav");

           Iterator<String> i=set.iterator();

           while(i.hasNext())

           {

           System.out.println(i.next());

           }

 }

}
```

```
Output:

Vijay
Ravi
Gaurav
Ajay
```

Java HashSet Example: Book

Let's see a HashSet example where we are adding books to set and printing all the books.

```
import java.util.*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;
}

}

public class HashSetExample {

public static void main(String[] args) {

    HashSet<Book> set=new HashSet<Book>();

    //Creating Books

    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

    //Adding Books to HashSet

    set.add(b1);

    set.add(b2);

    set.add(b3);

    //Traversing HashSet

    for(Book b:set){

    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

    }

}

}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4
```

1.5 LinkedHashSet Class

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.
- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.

Note: Keeping the insertion order in the LinkedHashset has some additional costs, both in terms of extra memory and extra CPU cycles. Therefore, if it is not required to maintain the insertion order, go for the lighter-weight HashMap or the HashSet instead.

Hierarchy of LinkedHashSet class

The LinkedHashSet class extends the HashSet class, which implements the Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

FileName: LinkedHashSet1.java

```
import java.util.*;
class LinkedHashSet1{
 public static void main(String args[]){
 //Creating HashSet and adding elements
        LinkedHashSet<String> set=new LinkedHashSet();
                set.add("One");
                set.add("Two");
                set.add("Three");
                set.add("Four");
                set.add("Five");
                Iterator<String> i=set.iterator();
                while(i.hasNext())
                {
                System.out.println(i.next());
                }
 }
}
```

Output:

```
One
Two
Three
Four
Five
```

Note: We can also use the enhanced for loop for displaying the elements.

Java LinkedHashSet example ignoring duplicate Elements

FileName: LinkedHashSet2.java

```
import java.util.*;
class LinkedHashSet2{
 public static void main(String args[]){
  LinkedHashSet<String> al=new LinkedHashSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
Ravi
Vijay
Ajay
```

Remove Elements Using LinkeHashSet Class

FileName: LinkedHashSet3.java

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

```
public class LinkedHashSet3

{


// main method

public static void main(String argvs[])

{


// Creating an empty LinekdhashSet of string type

LinkedHashSet<String> lhs = new LinkedHashSet<String>();


// Adding elements to the above Set

// by invoking the add() method

lhs.add("Java");

lhs.add("T");

lhs.add("Point");

lhs.add("Good");

lhs.add("Website");


// displaying all the elements on the console

System.out.println("The hash set is: " + lhs);


// Removing an element from the above linked Set


// since the element "Good" is present, therefore, the method remove()
// returns true
System.out.println(lhs.remove("Good"));

// After removing the element
System.out.println("After removing the element, the hash set is: " + lhs);

// since the element "For" is not present, therefore, the method remove()
// returns false
System.out.println(lhs.remove("For"));

}
}
```

Output:

```
The hash set is: [Java, T, Point, Good, Website]
true
After removing the element, the hash set is: [Java, T, Point, Website]
false
```

Java LinkedHashSet Example: Book

FileName: Book.java

```
import java.util.*;
class Book {
```

```
int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;

}

}

public class LinkedHashSetExample {

public static void main(String[] args) {

    LinkedHashSet<Book> hs=new LinkedHashSet<Book>();

    //Creating Books

    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);

    //Adding Books to hash table

    hs.add(b1);

    hs.add(b2);

    hs.add(b3);

    //Traversing hash table

    for(Book b:hs){
```

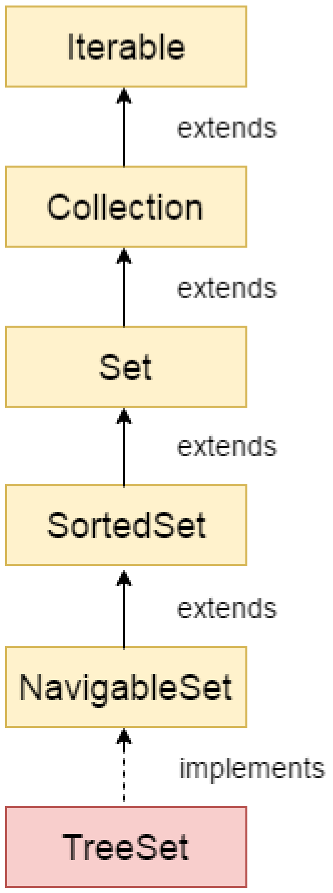
```
    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

1.6 TreeSet class

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.



The important points about the Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.
- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

Internal Working of The TreeSet Class

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

Hierarchy of TreeSet class

As shown in the above diagram, the Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

Java TreeSet Examples

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

FileName: TreeSet1.java

```
import java.util.*;
class TreeSet1{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java TreeSet Example 2:

Let's see an example of traversing elements in descending order.

FileName: TreeSet2.java

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

```
class TreeSet2{
 public static void main(String args[]){
 TreeSet<String> set=new TreeSet<String>();
         set.add("Ravi");
         set.add("Vijay");
         set.add("Ajay");
         System.out.println("Traversing element through Iterator in descending order");
         Iterator i=set.descendingIterator();
         while(i.hasNext())
         {
             System.out.println(i.next());
         }

 }
}
```

Output:

```
Traversing element through Iterator in descending order
Vijay
Ravi
```

```
Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay
```

Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

FileName: TreeSet3.java

```
import java.util.*;

class TreeSet3{

 public static void main(String args[]){

 TreeSet<Integer> set=new TreeSet<Integer>();

         set.add(24);

         set.add(66);

         set.add(12);

         set.add(15);

         System.out.println("Lowest Value: "+set.pollFirst());

         System.out.println("Highest Value: "+set.pollLast());

 }

}
```

```
Output:

Lowest Value: 12
Highest Value: 66
```

Java TreeSet Example: Book

Let's see a TreeSet example where we are adding books to the set and printing all the books. The elements in TreeSet must be of a Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement the Comparable interface.

FileName: TreeSetExample.java

```
import java.util.*;

class Book implements Comparable<Book>{

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;
```

```
   this.quantity = quantity;
}
// implementing the abstract method
public int compareTo(Book b) {
   if(id>b.id){
      return 1;
   }else if(id<b.id){
      return -1;
   }else{
   return 0;
   }
}
}
public class TreeSetExample {
public static void main(String[] args) {
   Set<Book> set=new TreeSet<Book>();
   //Creating Books
   Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
   Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
   Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);
   //Adding Books to TreeSet
   set.add(b1);
```

```
    set.add(b2);

    set.add(b3);

    //Traversing TreeSet

    for(Book b:set){

    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);

    }

}

}
```

Output:

```
101 Data Communications & Networking Forouzan Mc Graw Hill 4

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6
```

1.6 Queue Interface

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java. Implementations done by these classes are not thread safe. If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option.

Queue Interface Declaration

```
public interface Queue<E> extends Collection<E>
```

Features of a Queue

The following are some important features of a queue.

- As discussed earlier, FIFO concept is used for insertion and deletion of elements from a queue.
- The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.
- PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.
- The NullPointerException is raised, if any null operation is done on the BlockingQueues.
- Those Queues that are present in the *util* package are known as Unbounded Queues.
- Those Queues that are present in the *util.concurrent* package are known as bounded Queues.
- All Queues barring the Deques facilitates removal and insertion at the head and tail of the queue; respectively. In fact, deques support element insertion and removal at both ends.

PriorityQueue Class

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

Java PriorityQueue Example

FileName: TestCollection12.java

```
import java.util.*;
class TestCollection12{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

Output:

```
head:Amit
     head:Amit
     iterating the queue elements:
     Amit
```

```
Jai
Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay
```

Java PriorityQueue Example: Book

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

FileName: LinkedListExample.java

```
import java.util.*;

class Book implements Comparable<Book>{

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

   this.id = id;

   this.name = name;

   this.author = author;

   this.publisher = publisher;

   this.quantity = quantity;

}

public int compareTo(Book b) {
   if(id>b.id){

       return 1;

   }else if(id<b.id){

       return -1;

   }else{

   return 0;

   }

}

}

public class LinkedListExample {

public static void main(String[] args) {

   Queue<Book> queue=new PriorityQueue<Book>();

   //Creating Books

   Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);

   Book b2=new Book(233,"Operating System","Galvin","Wiley",6);

   Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);

   //Adding Books to the queue

   queue.add(b1);

   queue.add(b2);

   queue.add(b3);

   System.out.println("Traversing the queue elements:");

   //Traversing queue elements
   for(Book b:queue){
   System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
   }
   queue.remove();
   System.out.println("After removing one book record:");
   for(Book b:queue){
       System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
       }
}
}
```

Output:

```
Traversing the queue elements:
101 Data Communications & Networking Forouzan Mc Graw Hill 4
233 Operating System Galvin Wiley 6
121 Let us C Yashwant Kanetkar BPB 8
After removing one book record:
121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6
```

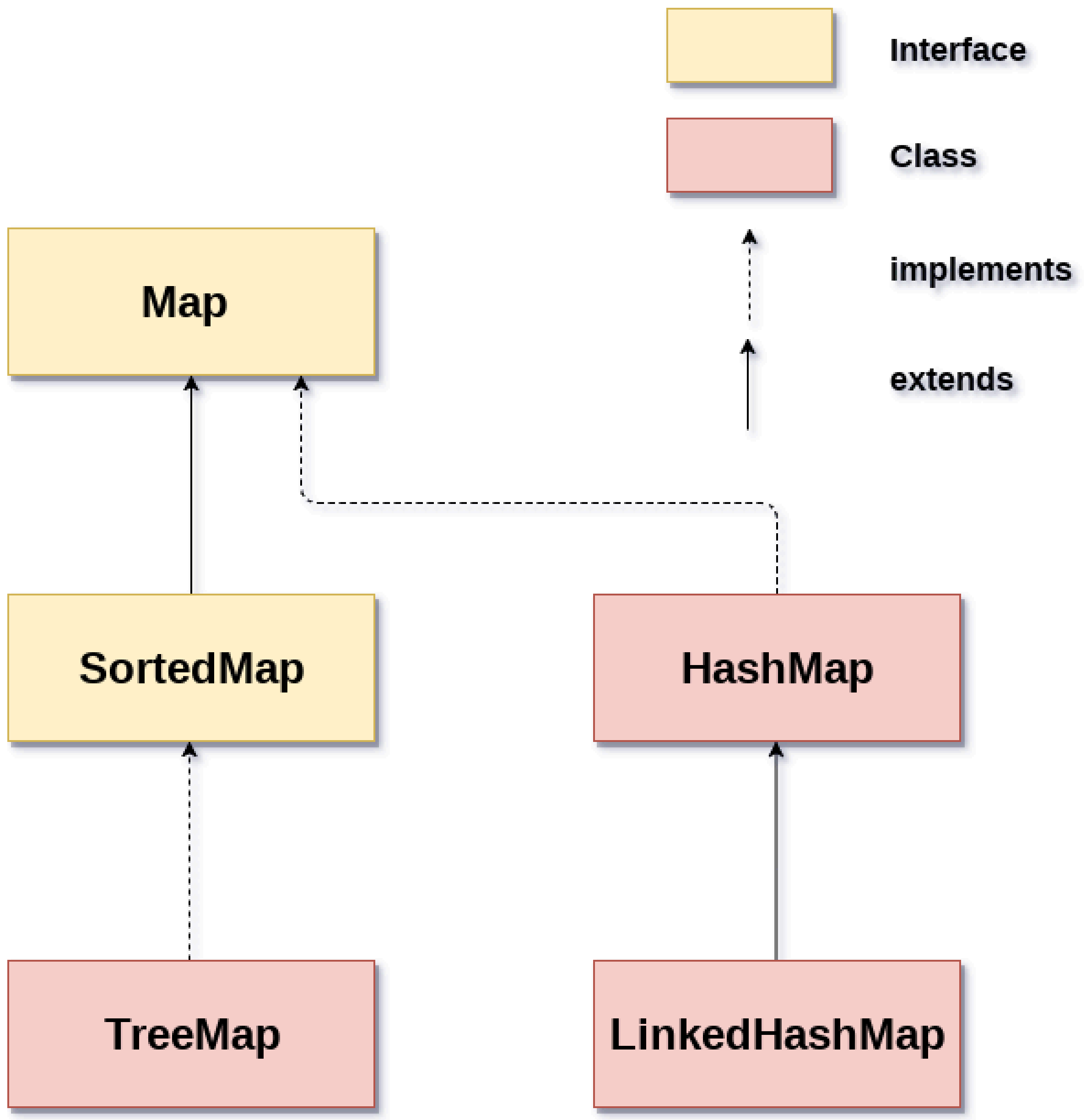
1.8 Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet()* method.

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

Java Map Example: Generic (New Style)

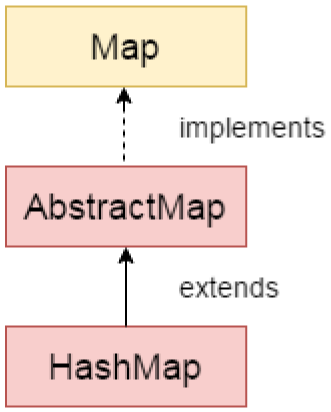
```
import java.util.*;
class MapExample2{
 public static void main(String args[]){
  Map<Integer,String> map=new HashMap<Integer,String>();
  map.put(100,"Amit");
  map.put(101,"Vijay");
  map.put(102,"Rahul");
  //Elements can traverse in any order
  for(Map.Entry m:map.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Output:

```
102 Rahul
100 Amit
101 Vijay
```

1.10 HashMap class

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.



It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.

Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

HashMap class Parameters

Let's see the Parameters for java.util.HashMap class.

- **K**: It is the type of keys maintained by this map.

- **V**: It is the type of mapped values.

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

```
import java.util.*;

public class HashMapExample1{

 public static void main(String args[]){

   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap

   map.put(1,"Mango");  //Put elements in Map

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(4,"Grapes");


   System.out.println("Iterating Hashmap...");

   for(Map.Entry m : map.entrySet()){

    System.out.println(m.getKey()+" "+m.getValue());

   }

}

}
```

Output:

```
Iterating Hashmap...
```

```
1 Mango

2 Apple

3 Banana

4 Grapes
```

In this example, we are storing Integer as the key and String as the value, so we are using HashMap<Integer,String> as the type. The put() method inserts the elements in the map.

To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the *getKey()* and *getValue()* methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry.

No Duplicate Key on HashMap

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

```
import java.util.*;

public class HashMapExample2{

 public static void main(String args[]){

   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap

   map.put(1,"Mango");  //Put elements in Map

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(1,"Grapes"); //trying duplicate key
   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

Output:

```
Iterating Hashmap...
1 Grapes
2 Apple
3 Banana
```

Java HashMap example to add() elements

Here, we see different ways to insert elements.

```
import java.util.*;
class HashMap1{
 public static void main(String args[]){
   HashMap<Integer,String> hm=new HashMap<Integer,String>();
```

```
  System.out.println("Initial list of elements: "+hm);
   hm.put(100,"Amit");
   hm.put(101,"Vijay");
   hm.put(102,"Rahul");

   System.out.println("After invoking put() method ");
   for(Map.Entry m:hm.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }

   hm.putIfAbsent(103, "Gaurav");
   System.out.println("After invoking putIfAbsent() method ");
   for(Map.Entry m:hm.entrySet()){
        System.out.println(m.getKey()+" "+m.getValue());
       }
   HashMap<Integer,String> map=new HashMap<Integer,String>();
   map.put(104,"Ravi");
   map.putAll(hm);
   System.out.println("After invoking putAll() method ");
   for(Map.Entry m:map.entrySet()){
        System.out.println(m.getKey()+" "+m.getValue());
       }
}
```

```
}
```

Output:

```
Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi
```

Java HashMap example to remove() elements

Here, we see different ways to remove elements.

```
import java.util.*;

public class HashMap2 {

  public static void main(String args[]) {

   HashMap<Integer,String> map=new HashMap<Integer,String>();

     map.put(100,"Amit");

     map.put(101,"Vijay");

     map.put(102,"Rahul");

     map.put(103, "Gaurav");

   System.out.println("Initial list of elements: "+map);

   //key-based removal

   map.remove(100);

   System.out.println("Updated list of elements: "+map);

   //value-based removal

   map.remove(101);

   System.out.println("Updated list of elements: "+map);

   //key-value pair based removal

   map.remove(102, "Rahul");

   System.out.println("Updated list of elements: "+map);

  }

}
```

```
Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}
```

Java HashMap example to replace() elements

Here, we see different ways to replace elements.

```
import java.util.*;

class HashMap3{

 public static void main(String args[]){

   HashMap<Integer,String> hm=new HashMap<Integer,String>();

      hm.put(100,"Amit");

      hm.put(101,"Vijay");

      hm.put(102,"Rahul");

      System.out.println("Initial list of elements:");

     for(Map.Entry m:hm.entrySet())

     {

        System.out.println(m.getKey()+" "+m.getValue());

     }

     System.out.println("Updated list of elements:");
```

```
    hm.replace(102, "Gaurav");
    for(Map.Entry m:hm.entrySet())
    {
      System.out.println(m.getKey()+" "+m.getValue());
    }
    System.out.println("Updated list of elements:");
    hm.replace(101, "Vijay", "Ravi");
    for(Map.Entry m:hm.entrySet())
    {
      System.out.println(m.getKey()+" "+m.getValue());
    }
    System.out.println("Updated list of elements:");
    hm.replaceAll((k,v) -> "Ajay");
    for(Map.Entry m:hm.entrySet())
    {
      System.out.println(m.getKey()+" "+m.getValue());
    }
 }
}
```

Output:

```
Initial list of elements:
```

```
100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay
```

Difference between HashSet and HashMap

HashSet contains only values whereas HashMap contains an entry(key and value).

Java HashMap Example: Book

```
import java.util.*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;

}

}

public class MapExample {

public static void main(String[] args) {

    //Creating map of Books

    Map<Integer,Book> map=new HashMap<Integer,Book>();

    //Creating Books

    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);

    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to map
    map.put(1,b1);
    map.put(2,b2);
    map.put(3,b3);

    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

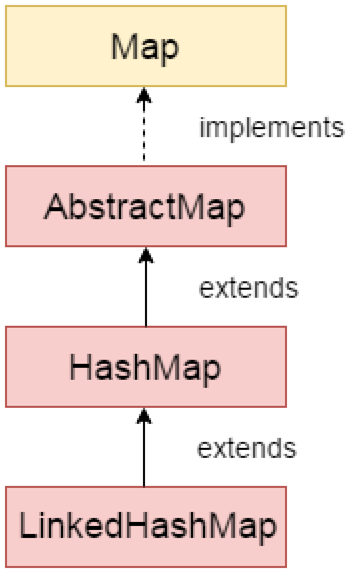
```
1 Details:
101 Let us C Yashwant Kanetkar BPB 8
2 Details:
102 Data Communications and Networking Forouzan Mc Graw Hill 4
```

3 Details:

103 Operating System Galvin Wiley 6

1.11 LinkedHashMap class

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.



Points to remember

- Java LinkedHashMap contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.

LinkedHashMap class declaration

Let's see the declaration for java.util.LinkedHashMap class.

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements
Map<K,V>
```

LinkedHashMap class Parameters

Let's see the Parameters for java.util.LinkedHashMap class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Java LinkedHashMap Example

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>();

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Output:

```
     100 Amit
     101 Vijay
```

```
102 Rahul
```

Java LinkedHashMap Example: Key-Value pair

```
import java.util.*;

class LinkedHashMap2{

 public static void main(String args[]){

   LinkedHashMap<Integer, String> map = new LinkedHashMap<Integer, String>();

      map.put(100,"Amit");

     map.put(101,"Vijay");

     map.put(102,"Rahul");

       //Fetching key

       System.out.println("Keys: "+map.keySet());

       //Fetching value

       System.out.println("Values: "+map.values());

       //Fetching key-value pair

       System.out.println("Key-Value pairs: "+map.entrySet());

 }

}
```

```
Output:

Keys: [100, 101, 102]

Values: [Amit, Vijay, Rahul]

Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]
```

Java LinkedHashMap Example:remove()

```
import java.util.*;

public class LinkedHashMap3 {

  public static void main(String args[]) {

   Map<Integer,String> map=new LinkedHashMap<Integer,String>();

    map.put(101,"Amit");

    map.put(102,"Vijay");

    map.put(103,"Rahul");

    System.out.println("Before invoking remove() method: "+map);

   map.remove(102);

   System.out.println("After invoking remove() method: "+map);

  }

}
```

```
Output:

Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}

After invoking remove() method: {101=Amit, 103=Rahul}
```

Java LinkedHashMap Example: Book

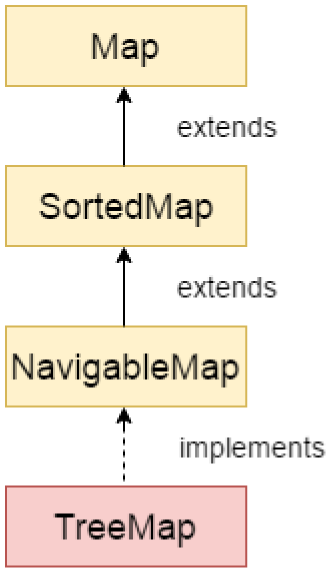
```
import java.util.*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;

    this.name = name;

    this.author = author;

    this.publisher = publisher;

    this.quantity = quantity;

}

}

public class MapExample {
```

```
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new LinkedHashMap<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to map
    map.put(2,b2);
    map.put(1,b1);
    map.put(3,b3);

    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

```
Output:


2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

3 Details:

103 Operating System Galvin Wiley 6
```

1.12 TreeMap class

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.



The important points about Java TreeMap class are:

- Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

TreeMap class Parameters

Let's see the Parameters for java.util.TreeMap class.

- **K**: It is the type of keys maintained by this map.
- **V**: It is the type of mapped values.

Java TreeMap Example

```
import java.util.*;

class TreeMap1{

 public static void main(String args[]){

   TreeMap<Integer,String> map=new TreeMap<Integer,String>();

      map.put(100,"Amit");

      map.put(102,"Ravi");

      map.put(101,"Vijay");

      map.put(103,"Rahul");
      for(Map.Entry m:map.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }
 }
}
```

```
Output:100 Amit
       101 Vijay
       102 Ravi
       103 Rahul
```

Java TreeMap Example: remove()

```
import java.util.*;
public class TreeMap2 {
   public static void main(String args[]) {
    TreeMap<Integer,String> map=new TreeMap<Integer,String>();
      map.put(100,"Amit");
      map.put(102,"Ravi");
      map.put(101,"Vijay");
      map.put(103,"Rahul");
      System.out.println("Before invoking remove() method");
      for(Map.Entry m:map.entrySet())
      {
          System.out.println(m.getKey()+" "+m.getValue());
      }
      map.remove(102);
      System.out.println("After invoking remove() method");
      for(Map.Entry m:map.entrySet())
      {
          System.out.println(m.getKey()+" "+m.getValue());
      }
      }
}
```

Output:

```
Before invoking remove() method
100 Amit
101 Vijay
102 Ravi
103 Rahul
After invoking remove() method
100 Amit
101 Vijay
103 Rahul
```

}

What is difference between HashMap and TreeMap?

HashMap	**TreeMap**
1) HashMap can contain one null key.	TreeMap cannot contain any null key.
2) HashMap maintains no order.	TreeMap maintains ascending order.

Java TreeMap Example: Book

```
import java.util.*;

class Book {

int id;

String name,author,publisher;

int quantity;

public Book(int id, String name, String author, String publisher, int quantity) {

   this.id = id;

   this.name = name;

   this.author = author;

   this.publisher = publisher;

   this.quantity = quantity;

}

}

public class MapExample {
```

```
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new TreeMap<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw
Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to map
    map.put(2,b2);
    map.put(1,b1);
    map.put(3,b3);

    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6
```

1.16 Collections class

Java collection class is used exclusively with static methods that operate on or return collections. It inherits the Object class.

Collections class declaration

Let's see the declaration for java.util.Collections class.

```
public class Collections extends Object
```

Java Collections Example

```
import java.util.*;

public class CollectionsExample {

    public static void main(String a[]){

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

        list.add("C");

        list.add("Core Java");

        list.add("Advance Java");

        System.out.println("Initial collection value:"+list);

        Collections.addAll(list, "Servlet","JSP");

        System.out.println("After adding elements collection value:"+list);

        String[] strArr = {"C#", ".Net"};

        Collections.addAll(list, strArr);

        System.out.println("After adding array collection value:"+list);

    }

}
```

Output:

```
Initial collection value:[C, Core Java, Advance Java]

After adding elements collection value:[C, Core Java, Advance Java, Servlet, JSP]
```

```
}
```

```
After adding array collection value:[C, Core Java, Advance Java, Servlet, JSP, C#, .Net]
```

Java Collections Example: max()

```
import java.util.*;

public class CollectionsExample {

    public static void main(String a[]){

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

        list.add(46);

        list.add(67);

        list.add(24);

        list.add(16);

        list.add(8);

        list.add(12);

        System.out.println("Value of maximum element from the collection: 
"+Collections.max(list));

    }

}
```

```
Output:

Value of maximum element from the collection: 67
```

Java Collections Example: min()

```
import java.util.*;

public class CollectionsExample {

    public static void main(String a[]){

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

        list.add(46);

        list.add(67);

        list.add(24);

        list.add(16);

        list.add(8);

        list.add(12);

        System.out.println("Value of minimum element from the collection: 
"+Collections.min(list));

    }

}
```

```
Output:

Value of minimum element from the collection: 8
```

Sorting in Collection

Collections class provides static methods for sorting the elements of a collection. If collection elements are of a Set type, we can use TreeSet. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): is used to sort the elements of List. List elements must be of the Comparable type.

> Note: String class and Wrapper classes implement the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example to sort string objects

```
import java.util.*;

class TestSort1{
```

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



ArrayList<String> al=new ArrayList<String>();

al.add("Viru");

al.add("Saurav");

al.add("Mukesh");

al.add("Tahir");



Collections.sort(al);

Iterator itr=al.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

 }

}

}
```

```
Output:

Mukesh

Saurav

Tahir

Viru
```

Example to sort string objects in reverse order

```
import java.util.*;

class TestSort2{

public static void main(String args[]){



ArrayList<String> al=new ArrayList<String>();

      al.add("Viru");

      al.add("Saurav");

      al.add("Mukesh");

      al.add("Tahir");



      Collections.sort(al,Collections.reverseOrder());
```

```
        Iterator i=al.iterator();

        while(i.hasNext())

        {

           System.out.println(i.next());

        }

}

}
```

```
Output:

Viru

Tahir

Saurav

Mukesh
```

1.19 Properties class

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

Recompilation is not required if the information is changed from a properties file: If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

Example of Properties class to get information from the properties file

To get information from the properties file, create the properties file first.

db.properties

```
user=system

password=oracle
```

Now, let's create the java class to read the data from the properties file.

Test.java

```
import java.util.*;

import java.io.*;

public class Test {

public static void main(String[] args)throws Exception{

   FileReader reader=new FileReader("db.properties");



   Properties p=new Properties();

   p.load(reader);



   System.out.println(p.getProperty("user"));

   System.out.println(p.getProperty("password"));

}

}
```

```
Output:system

     oracle
```

Now if you change the value of the properties file, you don't need to recompile the java class. That means no maintenance problem.

Example of Properties class to create the properties file

Now let's write the code to create the properties file.

Test.java

```
import java.util.*;

import java.io.*;

public class Test {

public static void main(String[] args)throws Exception{



Properties p=new Properties();

p.setProperty("name","Sonoo Jaiswal");

p.setProperty("email","sonoojaiswal@javatpoint.com");

p.store(new FileWriter("info.properties"),"Javatpoint Properties Example");

}

}
```

Let's see the generated properties file.

info.properties

```
#Javatpoint Properties Example
#Thu Oct 03 22:35:53 IST 2013
email=sonoojaiswal@javatpoint.com
name=Sonoo Jaiswal
```