Collections in Java

UNIT-II

A.Appandairaj

Utility Packages

Introduction to collection

Hierarchy of Collection framework

Generics, Array list, LL, HashSet, Treeset, HashMap

Comparators

Java annotations

Premain method.

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

- All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion, etc. can be achieved by Java Collections.

- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque, etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, etc.).

What is Collection in java

- A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interface.
- It is optional.

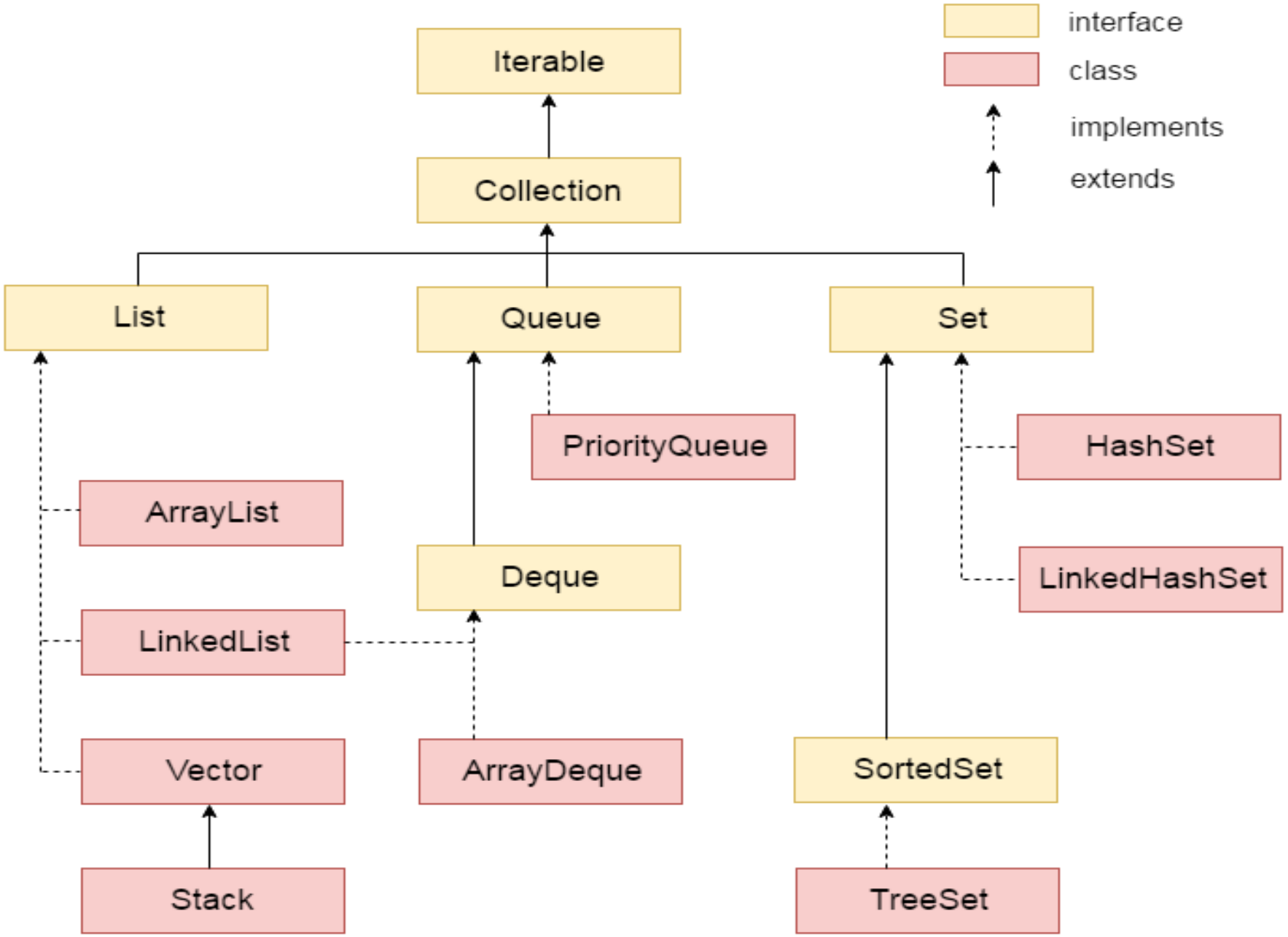
What is Collection framework

- The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:
- 1. Interfaces and its implementations, i.e., classes
- 2. Algorithm

Hierarchy of Collection Framework

- Let us see the hierarchy of Collection framework.

- The **java.util** package contains all the classes and interfaces for Collection framework.


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

Methods of Collection interface

- There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.

No.	Method	Description
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no. of elements from the collection.
8	public boolean contains(Object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collections.
14	public int hashCode()	returns the hash code number of the collection.

Methods of Iterator interface

- There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Java ArrayList class

- Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

- The important points about Java ArrayList class are:

1. Java ArrayList class can contain duplicate elements.

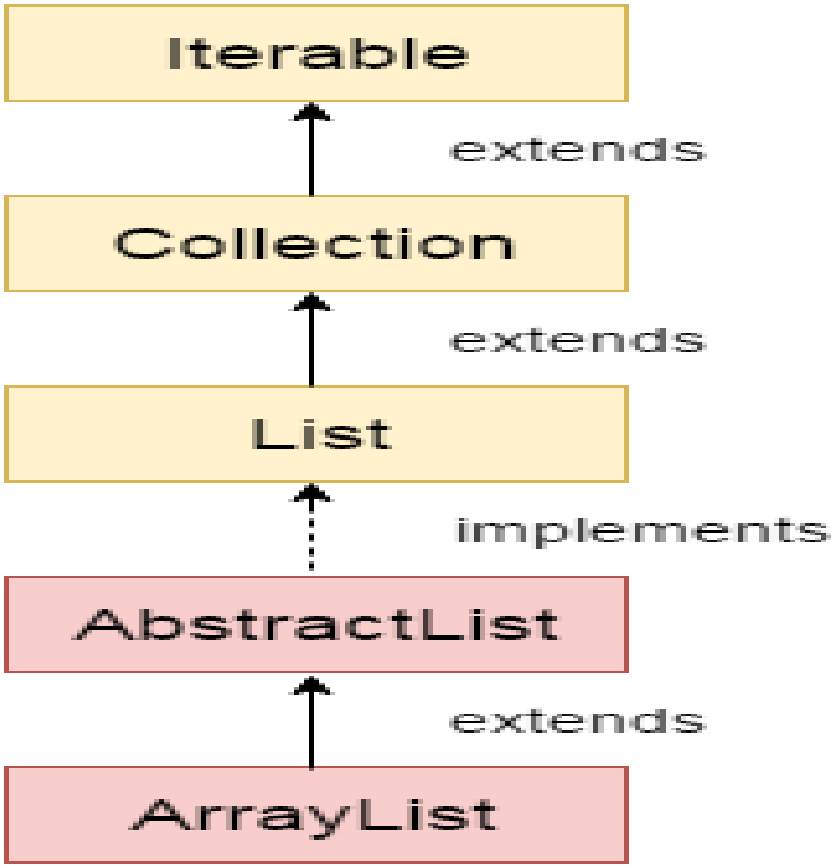
2. Java ArrayList class maintains insertion order.

3. Java ArrayList class is non synchronized.

4. Java ArrayList allows random access because array works at the index basis.

5. In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

Hierarchy of ArrayList class



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

ArrayList class declaration

- Let's see the declaration for java.util.ArrayList class.

- **public class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

Constructors of Java ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Methods of Java ArrayList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.

Method	Description
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

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 collection.

- Now it is type safe so typecasting is not required at run time.

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

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

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

- ArrayList<String> al=**new** ArrayList<String>();
- //creating new generic arraylist

- In generic collection, we specify the type in angular braces.

- Now ArrayList is forced to have only specified type of objects in it.

- If you try to add another type of object, it gives *compile time error*.

Java ArrayList Example

```
import java.util.*;
class TestCollection1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating array
     list
  list.add("Ravi");//Adding object in arraylist
  list.add("Vijay");
  list.add("Ravi");
  list.add("Ajay");
  //Traversing list through Iterator
  Iterator itr=list.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

Ravi

Vijay

Ravi

Ajay

Two ways to iterate the elements of collection in java

- There are two ways to traverse collection elements:
- 1. By Iterator interface.
- 2. By for-each loop.
- In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

Iterating Collection through for-each loop

```
import java.util.*;
class TestCollection2{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  for(String obj:al)
    System.out.println(obj);
 }
}
```

Output:
Ravi
Vijay
Ravi
Ajay

User-defined class objects in Java ArrayList

```
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.*;
public class TestCollection3{
 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

Example of addAll(Collection c)

```
import java.util.*;
class TestCollection4{
 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("Sonoo");
        al2.add("Hanumat");
        al.addAll(al2);//adding second list in first list
        Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:
Ravi
Vijay
Ajay
Sonoo
Hanumat

Example of removeAll()

```
import java.util.*;
class TestCollection5{
 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.removeAll(al2);
        System.out.println("iterating the elements after removing the elements of al2..."
           );
        Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }

  }
}
```

Output:
iterating the elements after removing the elements of al2...
Vijay
Ajay

Example of retainAll()

```
import java.util.*;
class TestCollection6{
 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 removing
the elements of al2...
Vijay
Ajay
```

Java ArrayList Example: Book

```
import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int quantity;
public Book(int id, String name, String author, String publish
  er, int quantity) {
      this.id = id;
      this.name = name;
      this.author = author;
      this.publisher = publisher;
      this.quantity = quantity;
}
}
```

```
public class ArrayListExample {
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 & Networking","Forouzan","M
          c 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:

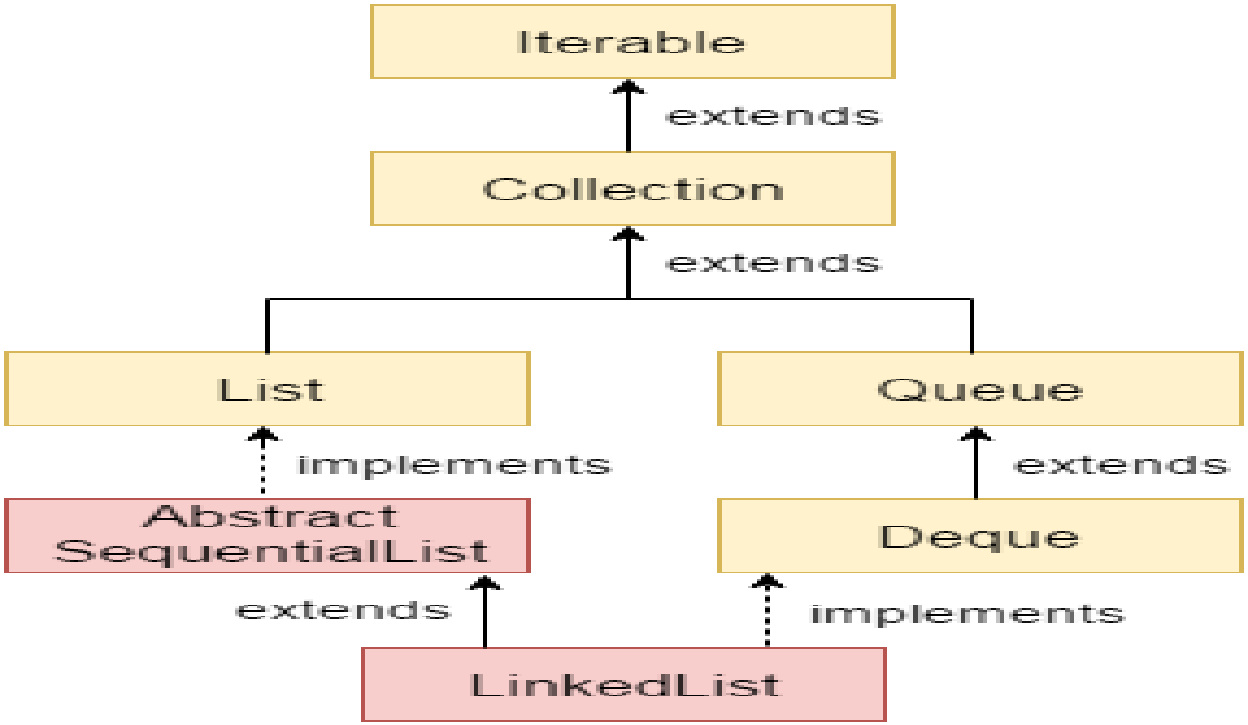
```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

Java LinkedList class

- Java LinkedList class uses 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 be occurred.
- Java LinkedList class can be used as list, stack or queue

Hierarchy of LinkedList class

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



Doubly Linked List

- In case of doubly linked list, we can add or remove elements from both side.

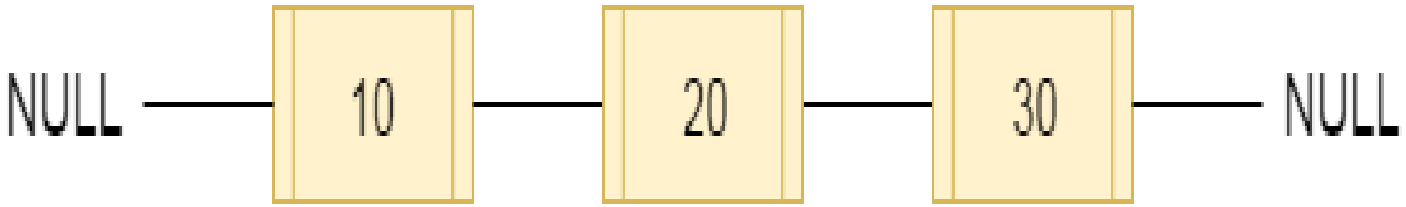


fig- doubly linked list

LinkedList class declaration

- Let's see the declaration for java.util.LinkedList class.
- **public class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection c)	It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Methods of Java LinkedList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.

Method	Description
oolean remove(Object o)	It is used to remove the first occurence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Java LinkedList Example

```
import java.util.*;
public class TestCollection7{
 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: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publis
her, 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	LinkedList
1) ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) 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.

Java 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.

Difference between List and Set

- List can contain duplicate elements whereas Set contains unique elements only.

Java HashSet class

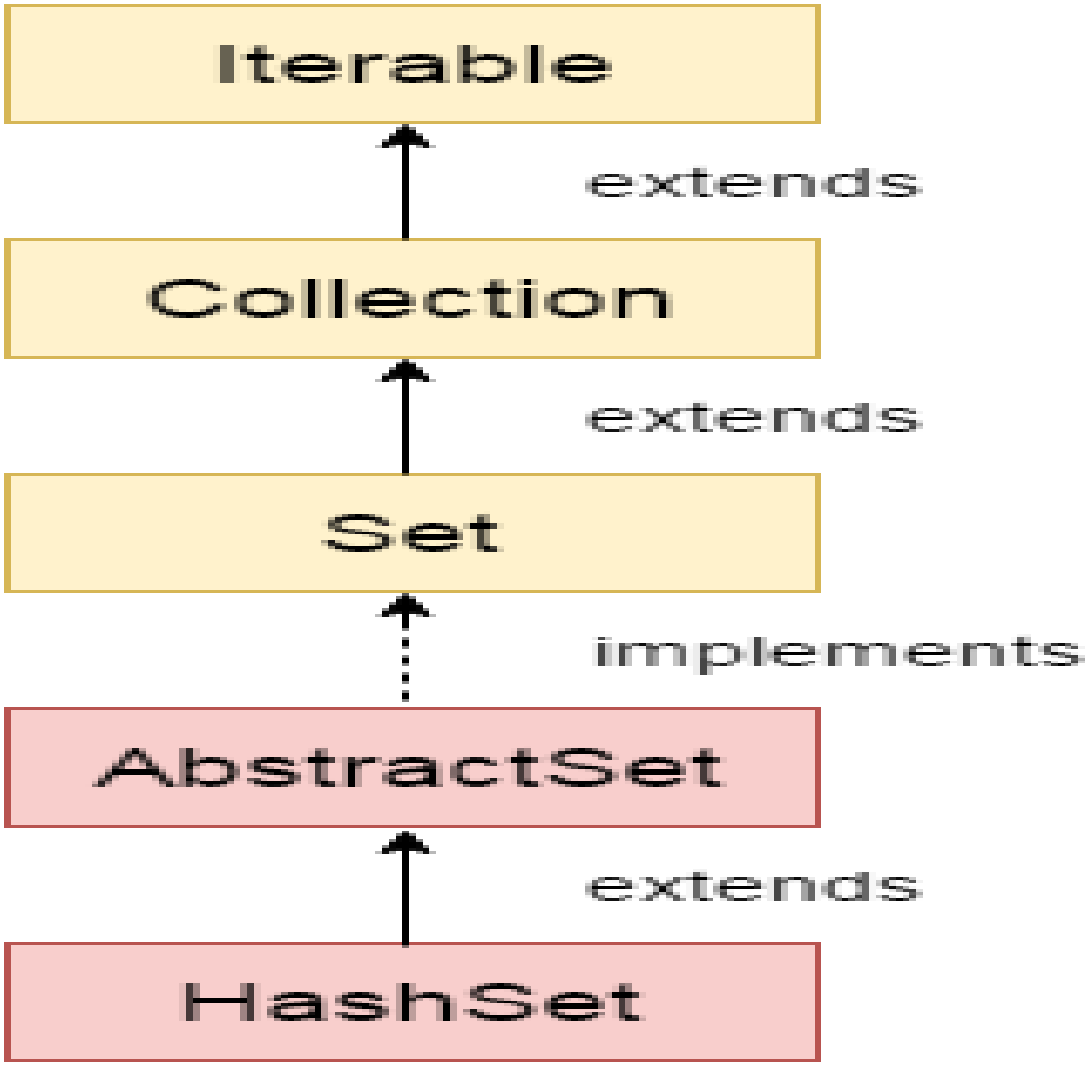
- Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:
- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.

Difference between List and Set

- 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.


Iterable
extends
Collection
extends
Set
implements
AbstractSet
extends
HashSet

HashSet class declaration

- Let's see the declaration for java.util.HashSet class.
- **public class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

Constructors of Java HashSet class

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.
2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.

Methods of Java HashSet class

SN	Modifier & Type	Method	Description
1)	boolean	add(Object o)	It is used to adds the specified element to this set if it is not already present.
2)	void	clear()	It is used to remove all of the elements from this set.
3)	object	clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
4)	boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator< E>	iterator()	It is used to return an iterator over the elements in this set.

SN	Modifier & Type	Method	Description
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in this set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in this set.

Java HashSet Example

```
import java.util.*;
class TestCollection9{
 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("Ravi1");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Output:

```
Ajay
Vijay
Ravi1
Ajay
```

Java HashSet Example: Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publis
her, 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:

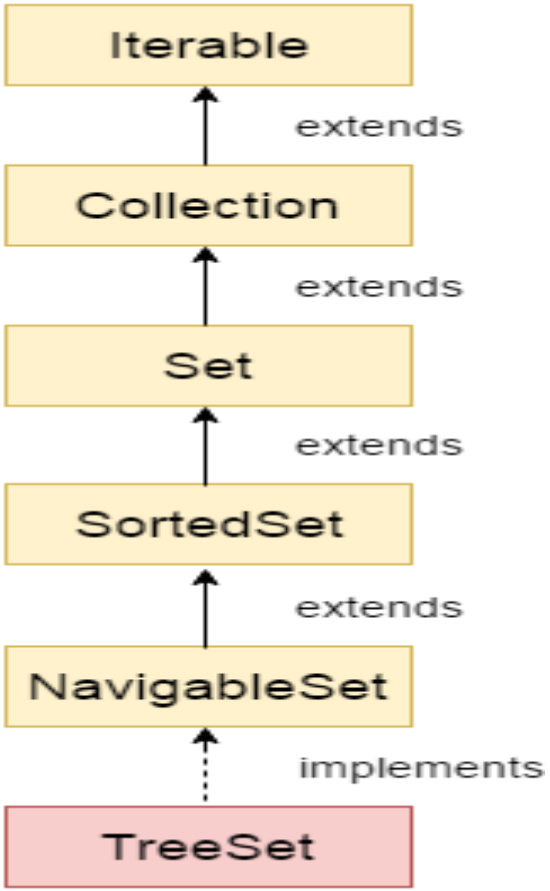
```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

Java TreeSet class

- Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.
- The important points about Java TreeSet class are:
- Contains unique elements only like HashSet.
- Access and retrieval times are quiet fast.
- Maintains ascending order.

Hierarchy of TreeSet class

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



TreeSet class declaration

- Let's see the declaration for java.util.TreeSet class.
- **public class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

Constructors of Java TreeSet class

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set.
TreeSet(Collection c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator comp)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet ss)	It is used to build a TreeSet that contains the elements of the given SortedSet.

Methods of Java TreeSet class

Method	Description
boolean addAll(Collection c)	It is used to add all of the elements in the specified collection to this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean isEmpty()	It is used to return true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void add(Object o)	It is used to add the specified element to this set if it is not already present.
void clear()	It is used to remove all of the elements from this set.

Method	Description
Object clone()	It is used to return a shallow copy of this TreeSet instance.
Object first()	It is used to return the first (lowest) element currently in this sorted set.
Object last()	It is used to return the last (highest) element currently in this sorted set.
int size()	It is used to return the number of elements in this set.

Java TreeSet Example

```
import java.util.*;
class TestCollection11{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi1");
  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: Book

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

```
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 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","Forouz
an","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:

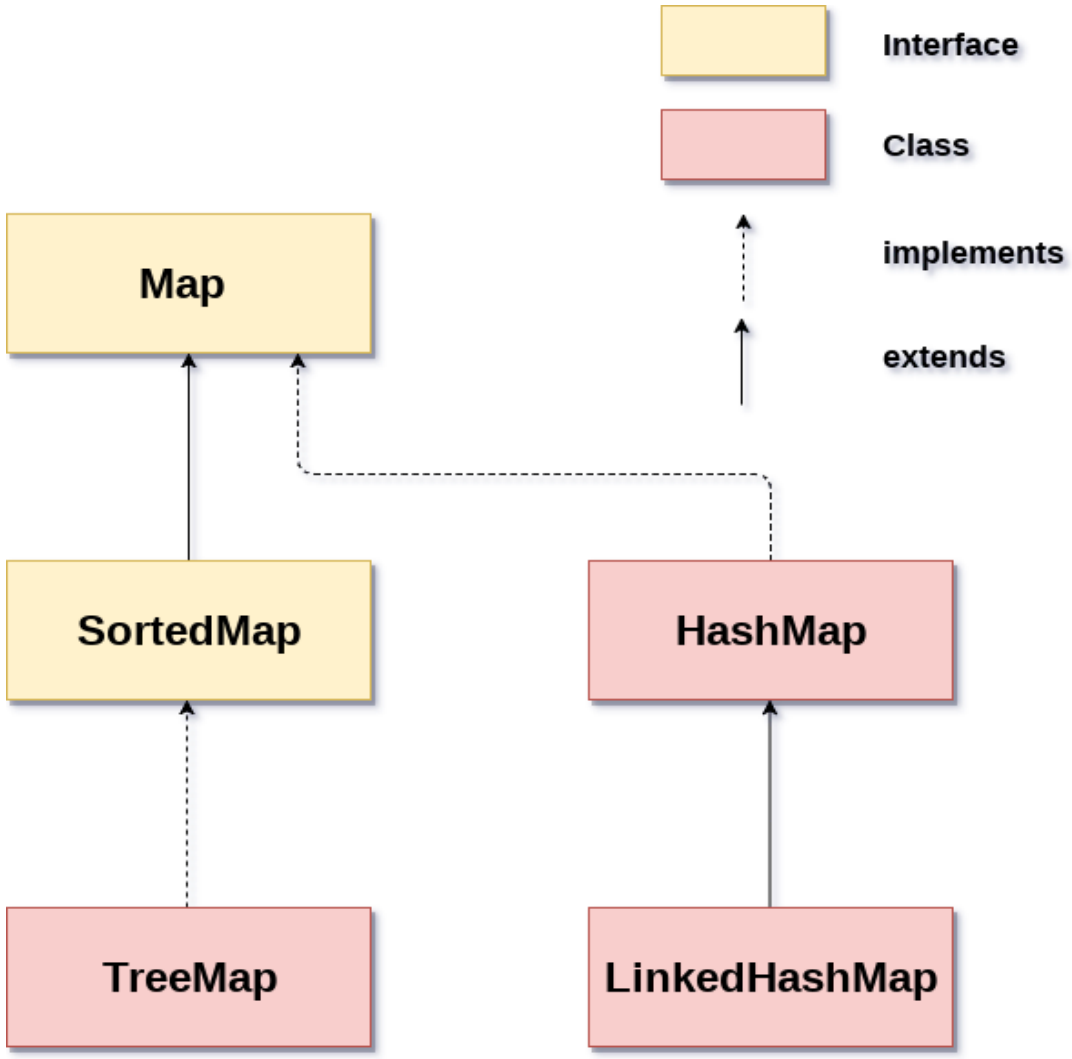
```
101 Data Communications & Networking Forouzan Mc Graw Hill 4
121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6
```

Java Map Interface

- Java 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. Map contains only unique keys.
- Map is useful if you have to search, update or delete elements on the basis of 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:



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

Useful methods of Map interface

Method	Description
Object put(Object key, Object value)	It is used to insert an entry in this map.
void putAll(Map map)	It is used to insert the specified map in this map.
Object remove(Object key)	It is used to delete an entry for the specified key.
Object get(Object key)	It is used to return the value for the specified key.
boolean containsKey(Object key)	It is used to search the specified key from this map.
Set keySet()	It is used to return the Set view containing all the keys.
Set entrySet()	It is used to return the Set view containing all the keys and values.

Map.Entry Interface

- Entry is the sub interface of Map. So we will be accessed it by Map.Entry name. It provides methods to get key and value.

Methods of Map.Entry interface

Method	Description
Object getKey()	It is used to obtain key.
Object getValue()	It is used to obtain value.

Java Map Example: Generic (New Style)

```
import java.util.*;
class MapInterfaceExample{
 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");
  for(Map.Entry m:map.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Output:
102 Rahul
100 Amit
101 Vijay

Java Map Example: Non-Generic (Old Style)

```
//Non-generic
import java.util.*;
public class MapExample1 {
public static void main(String[] args) {
    Map map=new HashMap();
    //Adding elements to map
    map.put(1,"Amit");
    map.put(5,"Rahul");
    map.put(2,"Jai");
    map.put(6,"Amit");
    //Traversing Map
    Set set=map.entrySet();//Converting to Set so that we can traverse
    Iterator itr=set.iterator();
    while(itr.hasNext()){
        //Converting to Map.Entry so that we can get key and value separately
        Map.Entry entry=(Map.Entry)itr.next();
        System.out.println(entry.getKey()+" "+entry.getValue());
    }
}
}
```

Output

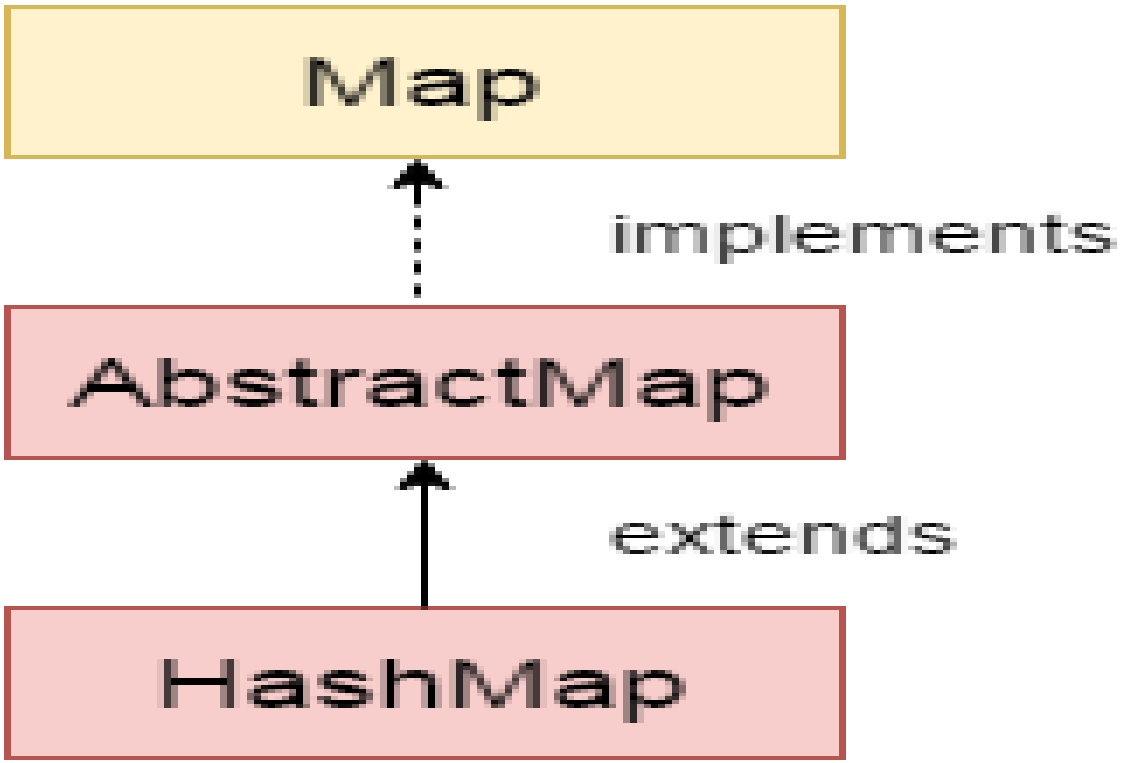
```
1 Amit
2 Jai
5 Rahul
6 Amit
```

Java HashMap class

- Java HashMap class implements the map interface by using a hashtable. It inherits AbstractMap class and implements Map interface.
- The important points about Java HashMap class are:
- A HashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It maintains no order

Hierarchy of HashMap class

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



HashMap class declaration

- Let's see the declaration for java.util.HashMap class.
- **public class** HashMap<K,V> **extends** Abstract Map<K,V> **implements** Map<K,V>, Cloneable, Serializable

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.

Constructors of Java HashMap class

Constructor	Description
HashMap()	It is used to construct a default HashMap.
HashMap(Map m)	It is used to initializes the hash map by using the elements of the given Map object m.
HashMap(int capacity)	It is used to initializes the capacity of the hash map to the given integer value, capacity.
HashMap(int capacity, float fillRatio)	It is used to initialize both the capacity and fill ratio of the hash map by using its arguments.

Methods of Java HashMap class

Method	Description
void clear()	It is used to remove all of the mappings from this map.
boolean containsKey(Object key)	It is used to return true if this map contains a mapping for the specified key.
boolean containsValue(Object value)	It is used to return true if this map maps one or more keys to the specified value.
boolean isEmpty()	It is used to return true if this map contains no key-value mappings.
Object clone()	It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

Method	Description
Set entrySet()	It is used to return a collection view of the mappings contained in this map.
Set keySet()	It is used to return a set view of the keys contained in this map.
Object put(Object key, Object value)	It is used to associate the specified value with the specified key in this map.
int size()	It is used to return the number of key-value mappings in this map.
Collection values()	It is used to return a collection view of the values contained in this map.

Java HashMap Example

```
import java.util.*;
class TestCollection13{
 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");
  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Output:

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

Java HashMap Example: remove()

```
import java.util.*;
public class HashMapExample {
   public static void main(String args[]) {
   // create and populate hash map
   HashMap<Integer, String> map = new HashMap<Integer, String>();
    map.put(101,"Let us C");
    map.put(102, "Operating System");
    map.put(103, "Data Communication and Networking");
    System.out.println("Values before remove: "+ map);
    // Remove value for key 102
    map.remove(102);
    System.out.println("Values after remove: "+ map);
    }
}
```

Output

Values before remove: {102=Operating System, 103=Data Communication and Networking, 101=Let us C}

Values after remove: {103=Data Communication and Networking, 101=Let us C}

Difference between HashSet and HashMap

- HashSet contains only values whereas HashMap contains 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 quantit
y) {
    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 & Networking Forouzan Mc Graw Hill 4
3 Details:
103 Operating System Galvin Wiley 6
```

Java Comparable interface

- Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on based on single data member only. For example it may be rollno, name, age or anything else.

compareTo(Object obj) method

- **public int compareTo(Object obj):** is used to compare the current object with the specified object.
- We can sort the elements of:
- String objects
- Wrapper class objects
- User-defined class objects

Collections class

- **Collections** class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But 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 Comparable type.

Java Comparable Example

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

public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
```

```
import java.util.*;
import java.io.*;
public class TestSort3{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

Output:
105 Jai 21
101 Vijay 23
106 Ajay 27

Java Comparator interface

- **Java Comparator interface** is used to order the objects of user-defined class.
- This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).
- It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

compare() method

- **public int compare(Object obj1,Object obj2):** compares the first object with second object.

Collections class

- **Collections** class provides static methods for sorting the elements of collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

Method of Collections class for sorting List elements

- **public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

Java Comparator Example (Non-generic Old Style)

- Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:
- Student.java
- AgeComparator.java
- NameComparator.java
- Simple.java

Student.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;
- }
- }

AgeComparator.java

- This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2 , 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

```
import java.util.*;
class AgeComparator implements Comparator{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java

- This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```
import java.util.*;
class NameComparator implements Comparat
or{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

return s1.name.compareTo(s2.name);
}
}
```

Simple.java

- In this class, we are printing the objects values by sorting on the basis of name and age.

```
import java.util.*;
import java.io.*;
  class Simple{
public static void main(String args[]){
  ArrayList al=new ArrayList();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));
  System.out.println("Sorting by Name...");
 Collections.sort(al,new NameComparator());
Iterator itr=al.iterator();
while(itr.hasNext()){
Student st=(Student)itr.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
 System.out.println("sorting by age...");
 Collections.sort(al,new AgeComparator());
Iterator itr2=al.iterator();
while(itr2.hasNext()){
Student st=(Student)itr2.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

OUTPUT

- Sorting by Name...
- 106 Ajay 27
- 105 Jai 21
- 101 Vijay 23
- Sorting by age...
- 105 Jai 21
- 101 Vijay 23
- 106 Ajay 27

Java Comparator Example (Generic)

Student.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;
}
}
```

AgeComparator.java

```
import java.util.*;
class AgeComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java

- This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.
- **import** java.util.*;
- **class** NameComparator **implements** Comparator<Stud ent>{
- **public int** compare(Student s1,Student s2){
- **return** s1.name.compareTo(s2.name);
- }
- }

Simple.java

- In this class, we are printing the objects values by sorting on the basis of name and age.

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

ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name...");

Collections.sort(al,new NameComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}

System.out.println("sorting by age...");

Collections.sort(al,new AgeComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}

}
}
```

Output:

```
Sorting by Name...
106 Ajay 27
105 Jai 21
101 Vijay 23

Sorting by age...
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Difference between Comparable and Comparator

Comparable	Comparator
1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc.	Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc.
2) Comparable **affects the original class** i.e. actual class is modified.	Comparator **doesn't affect the original class** i.e. actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is found in **java.lang** package.	Comparator is found in **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method.

Java **Annotations**

- *Annotations*, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, among them:
- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at runtime.
- This lesson explains where annotations can be used, how to apply annotations, what predefined annotation types are available in the Java Platform, Standard Edition (Java SE API), how type annnotations can be used in conjunction with pluggable type systems to write code with stronger type checking, and how to implement repeating annotations.

Annotations Basics
The Format of an Annotation

- In its simplest form, an annotation looks like the following:

  @Entity
- The at sign character (@) indicates to the compiler that what follows is an annotation. In the following example, the annotation's name is Override:

  @Override void mySuperMethod() { ... }
- The annotation can include *elements*, which can be named or unnamed, and there are values for those elements:

  @Author(name = "Benjamin Franklin", date = "3/27/2003") class MyClass() { ... }

  Or

  @SuppressWarnings(value = "unchecked") void myMethod() { ... }
- If there is just one element named value, then the name can be omitted, as in:
- @SuppressWarnings("unchecked") void myMethod() { ... }

- If the annotation has no elements, then the parentheses can be omitted, as shown in the previous @Override example.
- It is also possible to use multiple annotations on the same declaration:
- @Author(name = "Jane Doe") @EBook class MyClass { ... }
- If the annotations have the same type, then this is called a repeating annotation:
- @Author(name = "Jane Doe") @Author(name = "John Smith") class MyClass { ... }
- Repeating annotations are supported as of the Java SE 8 release. For more information, see Repeating Annotations.
- The annotation type can be one of the types that are defined in the java.lang or java.lang.annotation packages of the Java SE API. In the previous
examples, Override and SuppressWarnings are predefined Java annotations. It is also possible to define your own annotation type. The Author and Ebook annotations in the previous example are custom annotation types.

Where Annotations Can Be Used

- Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements. When used on a declaration, each annotation often appears, by convention, on its own line.
- As of the Java SE 8 release, annotations can also be applied to the *use* of types. Here are some examples:
- Class instance creation expression: new @Interned MyObject();
- Type cast: myString = (@NonNull String) str;
- implements clause: class UnmodifiableList<T> implements @Readonly List<@Readonly T> { ... }
- Thrown exception declaration: void monitorTemperature() throws @Critical TemperatureException { ... }
- This form of annotation is called a *type annotation*. For more information, see Type Annotations and Pluggable Type Systems.