UNIT-4

Collections in Java

Collections 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 performed by Java Collections.

Java Collection simply 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

Collection represents a single unit of objects i.e. a group.

What is framework in java

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

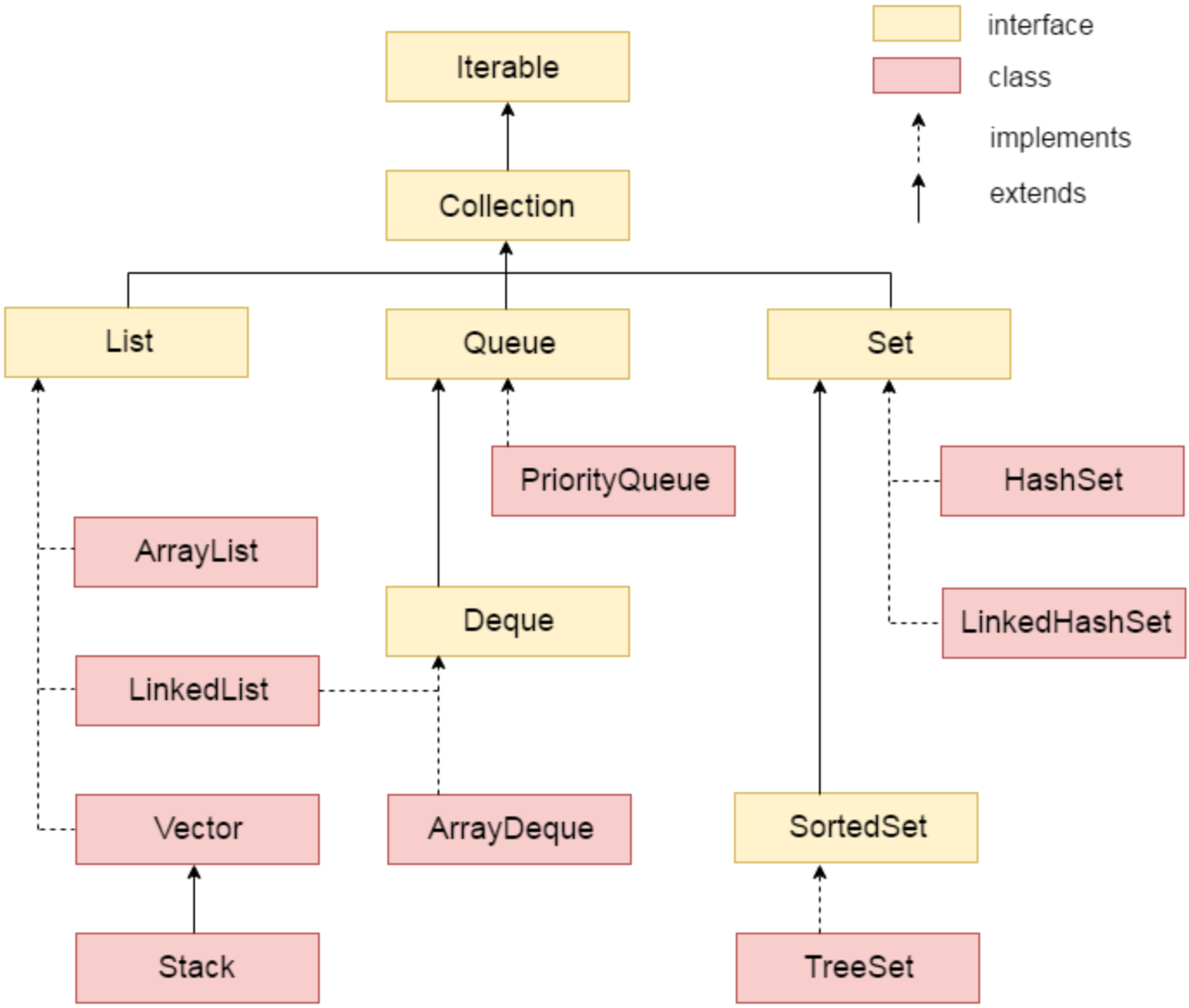
What is Collection framework

Collection framework represents a unified architecture for storing and manipulating 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.



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	is used to delete an element from this collection.

	element)	
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.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element 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 collection.
14	public int hashCode()	returns the hashcode number for collection.

Iterator interface

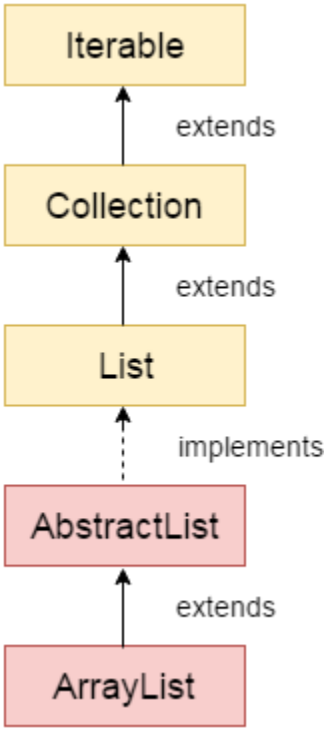
Iterator interface provides the facility of iterating the elements in forward direction only.

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 iterator has more elements.
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 rarely 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:

- 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 array works at the index basis.

- 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.

1. **public class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, S erializable

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.
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.

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

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

```
1. 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*.

For more information of java generics, click here Java Generics Tutorial.

Java ArrayList Example

```
import java.util.*;
class TestCollection1{
 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");
  //Traversing list through Iterator
  Iterator itr=list.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
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);
 }
}
```

```
Ravi
Vijay
Ravi
Ajay
```

User-defined class objects in Java ArrayList

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

```
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);
  }
 }
}
```

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

Example of addAll(Collection c) method

```
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());
  }
 }
}
```

```
Ravi
Vijay
Ajay
Sonoo
Hanumat
```

Example of removeAll() method

```
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());
  }

  }
```

```
}
 iterating the elements after removing the elements of al2...
 Vijay
 Ajay
```

Example of retainAll() method

```
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());
  }
 }
}
```

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

Java ArrayList Example: Book

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

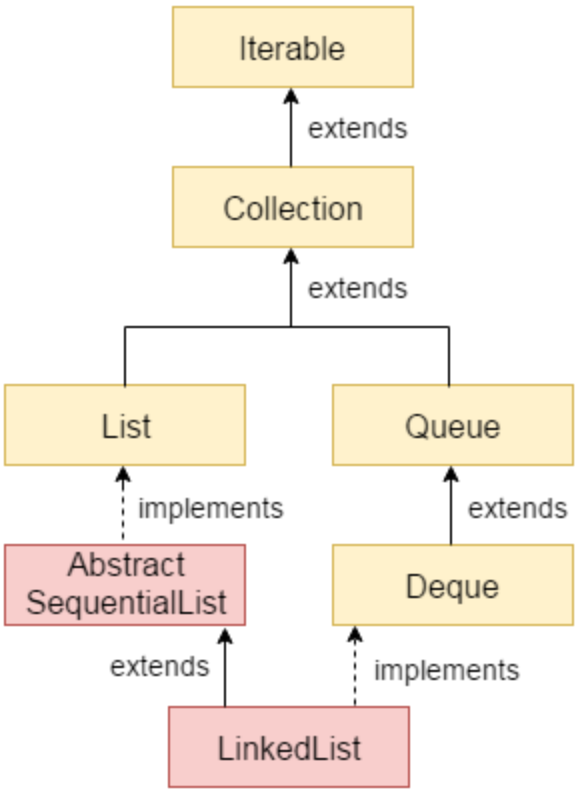
```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}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

```
}
public class 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","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
```

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.

1. **public class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Clone able, 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.
boolean 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 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 implements List interface and maintains insertion order. Both are non synchronized classes.

But there are many differences between ArrayList and LinkedList classes that are given below.

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 List Interface

List Interface is the subinterface of Collection.It contains methods to insert and delete elements in index basis.It is a factory of ListIterator interface.

List Interface declaration

1. **public interface** List<E> **extends** Collection<E>
2.

Methods of Java List Interface

Method	Description
void add(int index,Object element)	It is used to insert element into the invoking list at the index passed in the index.
boolean addAll(int index,Collection c)	It is used to insert all elements of c into the invoking list at the index passed in the index.
object get(int index)	It is used to return the object stored at the specified index within the invoking collection.
object set(int index,Object element)	It is used to assign element to the location specified by index within the invoking list.
object remove(int index)	It is used to remove the element at position index from the invoking list and return the deleted element.
ListIterator listIterator()	It is used to return an iterator to the start of the invoking list.
ListIterator listIterator(int index)	It is used to return an iterator to the invoking list that begins at the specified index.

Java List Example

```
import java.util.*;
public class ListExample{
public static void main(String args[]){
ArrayList<String> al=new ArrayList<String>();
al.add("Amit");
al.add("Vijay");
al.add("Kumar");
al.add(1,"Sachin");
System.out.println("Element at 2nd position: "+al.get(2));
for(String s:al){
 System.out.println(s);
}
}
}
```

Output:

```
Element at 2nd position: Vijay
Amit
Sachin
Vijay
Kumar
```

Java ListIterator Interface

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

ListIterator Interface declaration

1. **public interface** ListIterator<E> **extends** Iterator<E>

Methods of Java ListIterator Interface:

Method	Description
boolean hasNext()	This method return true if the list iterator has more elements when traversing the list in the forward direction.
Object next()	This method return the next element in the list and advances the cursor position.
boolean hasPrevious()	This method return true if this list iterator has more elements when traversing the list in the reverse direction.

Object previous()	This method return the previous element in the list and moves the cursor position backwards.

Example of ListIterator Interface

```
import java.util.*;
public class TestCollection8{
public static void main(String args[]){
ArrayList<String> al=new ArrayList<String>();
al.add("Amit");
al.add("Vijay");
al.add("Kumar");
al.add(1,"Sachin");
System.out.println("element at 2nd position: "+al.get(2));
ListIterator<String> itr=al.listIterator();
System.out.println("traversing elements in forward direction...");
while(itr.hasNext()){
System.out.println(itr.next());
}
System.out.println("traversing elements in backward direction...");
while(itr.hasPrevious()){
System.out.println(itr.previous());
}
}
}
```

Output:

```
element at 2nd position: Vijay
traversing elements in forward direction...
Amit
Sachin
Vijay
Kumar
traversing elements in backward direction...
Kumar
Vijay
Sachin
Amit
```

Example of ListIterator Interface: Book

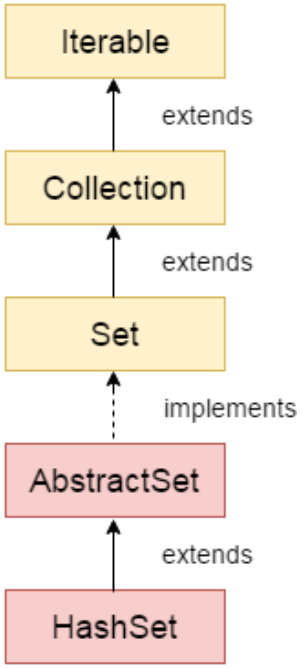
```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
```

```
    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 ListExample {
    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","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
```

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.

HashSet class declaration

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

1. **public class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

Constructors of Java HashSet class:

Constructor	Description

HashSet()	It is used to construct a default HashSet.
HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.
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.

Methods of Java HashSet class:

Method	Description
void clear()	It is used to remove all of the elements from this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean add(Object o)	It is used to adds the specified element to this set if it is not already present.
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.
Object clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
Iterator iterator()	It is used to return an iterator over the elements in this set.
int size()	It is used to return the number of 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("Ravi");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Ajay
Vijay
Ravi
```

Java HashSet Example: Book

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

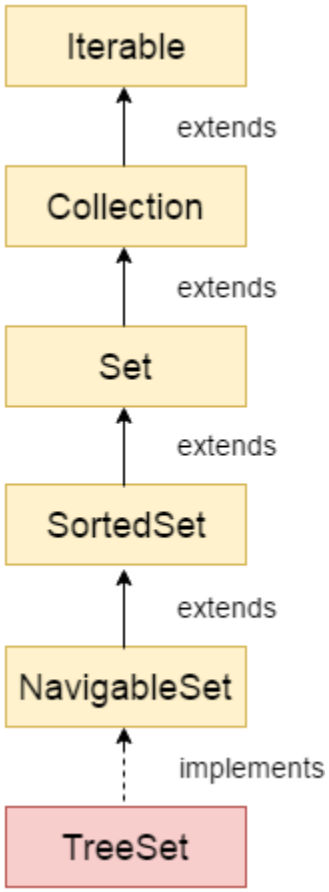
```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}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

1. **public class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializabl
   e

Constructors of Java TreeSet class

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in an ascending orde 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 co
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.
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("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: 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","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
```

Java Queue Interface

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

Queue Interface declaration

1. **public interface** Queue<E> **extends** Collection<E>

Methods of Java Queue Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PriorityQueue class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

PriorityQueue class declaration

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

1. **public class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

Java PriorityQueue Example

```
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.

```
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
```

Java Deque Interface

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

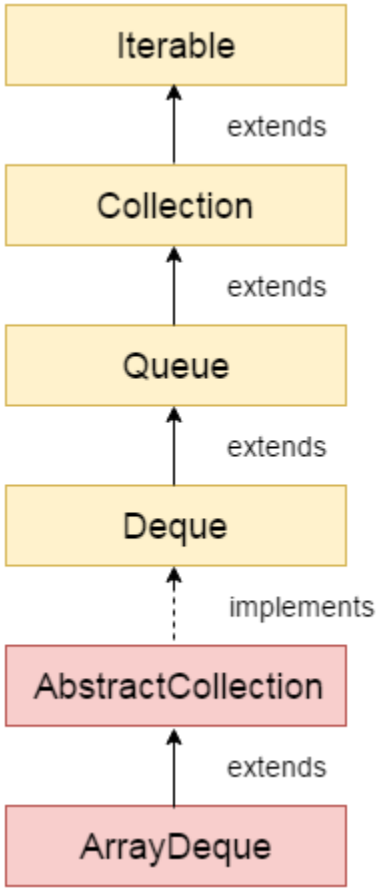
Deque Interface declaration

1. **public interface** Deque<E> **extends** Queue<E>

Methods of Java Deque Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this deque and return true upon success.
boolean offer(object)	It is used to insert the specified element into this deque.
Object remove()	It is used to retrieves and removes the head of this deque.

Object poll()	It is used to retrieves and removes the head of this deque, or returns null if this deque is empty.
Object element()	It is used to retrieves, but does not remove, the head of this deque.
Object peek()	It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty.



ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.

- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

ArrayDeque Hierarchy

The hierarchy of ArrayDeque class is given in the figure displayed at the right side of the page.

ArrayDeque class declaration

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

1. **public class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Seriali zable

Java ArrayDeque Example

```
import java.util.*;
public class ArrayDequeExample {
  public static void main(String[] args) {
  //Creating Deque and adding elements
  Deque<String> deque = new ArrayDeque<String>();
  deque.add("Ravi");
  deque.add("Vijay");
  deque.add("Ajay");
  //Traversing elements
  for (String str : deque) {
  System.out.println(str);
  }
  }
}
```

Output:

```
Ravi
Vijay
Ajay
```

Java ArrayDeque Example: offerFirst() and pollLast()

```
import java.util.*;
public class DequeExample {
public static void main(String[] args) {
   Deque<String> deque=new ArrayDeque<String>();
   deque.offer("arvind");
   deque.offer("vimal");
   deque.add("mukul");
```

```
        deque.offerFirst("jai");
        System.out.println("After offerFirst Traversal...");
        for(String s:deque){
            System.out.println(s);
        }
        //deque.poll();
        //deque.pollFirst();//it is same as poll()
        deque.pollLast();
        System.out.println("After pollLast() Traversal...");
        for(String s:deque){
            System.out.println(s);
        }
    }
    }
```

Output:

```
After offerFirst Traversal...
jai
arvind
vimal
mukul
After pollLast() Traversal...
jai
arvind
vimal
```

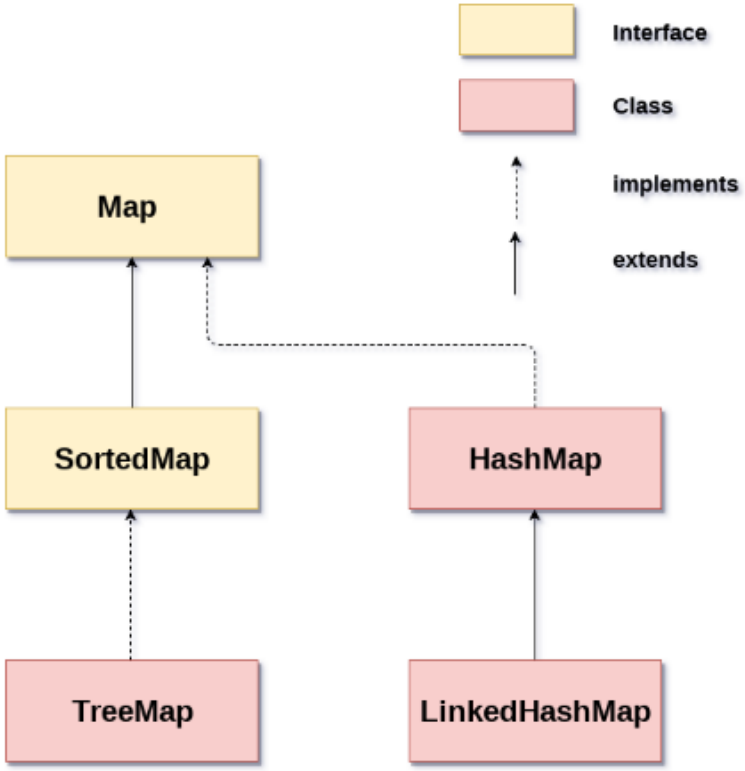
Java Map Interface

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:



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.

Useful methods of Map interface

Method	Description
Object put(Object key, Object	It is used to insert an entry in this map.

value)	
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
```

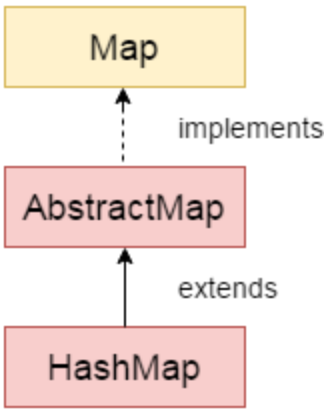
Java Map Example: Non-Generic (Old Style)

```
//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.

1. **public class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializabl e

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.
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

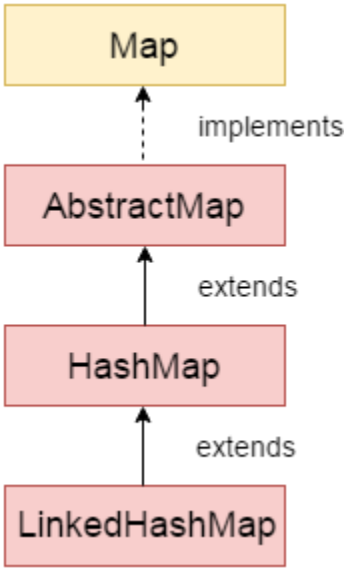
```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
}
public 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 & Networking Forouzan Mc Graw Hill 4
3 Details:
103 Operating System Galvin Wiley 6
```

Java LinkedHashMap class



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

The important points about Java LinkedHashMap class are:

- A LinkedHashMap contains values based on the key.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is same as HashMap instead maintains insertion order.

LinkedHashMap class declaration

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

1. **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.

Constructors of Java LinkedHashMap class

Constructor	Description
LinkedHashMap()	It is used to construct a default LinkedHashMap.
LinkedHashMap(int capacity)	It is used to initialize a LinkedHashMap with the given capacity.
LinkedHashMap(int capacity, float fillRatio)	It is used to initialize both the capacity and the fillRatio.
LinkedHashMap(Map m)	It is used to initialize the LinkedHashMap with the elements from the given Map class m.

Methods of Java LinkedHashMap class

Method	Description
Object get(Object key)	It is used to return the value to which this map maps the specified key.
void clear()	It is used to remove all mappings from this map.
boolean containsKey(Object key)	It is used to return true if this map maps one or more keys to the specified value.

Java LinkedHashMap Example

```
import java.util.*;
class TestCollection14{
 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
```

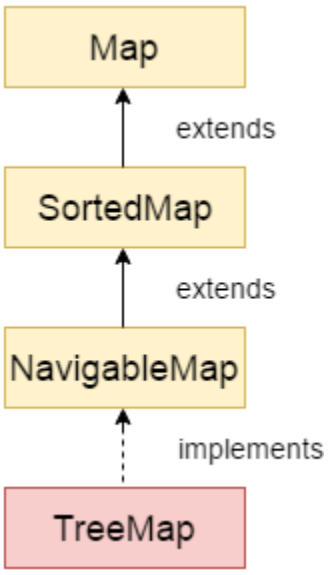
Java LinkedHashMap Example:remove()

```
import java.util.*;
public class LinkedHashMapExample {
  public static void main(String args[]) {
  // Create and populate linked hash map
  Map<Integer, String> map = new LinkedHashMap<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: {101=Let us C, 102=Operating System, 103=Data Communication and
Networking}
Values after remove: {101=Let us C, 103=Data Communication and Networking}
```

Java TreeMap class



Java TreeMap class implements the Map interface by using a tree. It provides an efficient means of storing key/value pairs in sorted order.

The important points about Java TreeMap class are:

- A TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- It contains only unique elements.
- It cannot have null key but can have multiple null values.
- It is same as HashMap instead maintains ascending order.

TreeMap class declaration

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

1. **public class** TreeMap<K,V> **extends** AbstractMap<K,V> **implements** NavigableMap<K,V>, Cloneable, S erializable

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.

Constructors of Java TreeMap class

Constructor	Description
TreeMap()	It is used to construct an empty tree map that will be sorted using the natural order of its key.
TreeMap(Comparator comp)	It is used to construct an empty tree-based map that will be sorted using the comparator comp.
TreeMap(Map m)	It is used to initialize a tree map with the entries from **m**, which will be sorted using the natural order of the keys.
TreeMap(SortedMap sm)	It is used to initialize a tree map with the entries from the SortedMap **sm**, which will be sorted in the same order as **sm.**

Methods of Java TreeMap class

Method	Description
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.
Object firstKey()	It is used to return the first (lowest) key currently in this sorted map.
Object get(Object key)	It is used to return the value to which this map maps the specified key.
Object lastKey()	It is used to return the last (highest) key currently in this sorted map.
Object remove(Object key)	It is used to remove the mapping for this key from this TreeMap if present.
void putAll(Map map)	It is used to copy all of the mappings from the specified map to this map.
Set entrySet()	It is used to return a set view of the mappings contained 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 TreeMap Example:

```
import java.util.*;
class TestCollection15{
 public static void main(String args[]){
```

```
        TreeMap<Integer,String> hm=new TreeMap<Integer,String>();
        hm.put(100,"Amit");
        hm.put(102,"Ravi");
        hm.put(101,"Vijay");
        hm.put(103,"Rahul");
        for(Map.Entry m:hm.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 TreeMapExample {
   public static void main(String args[]) {
   // Create and populate tree map
   Map<Integer, String> map = new TreeMap<Integer, String>();
   map.put(102,"Let us C");
   map.put(103, "Operating System");
   map.put(101, "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: {101=Data Communication and Networking, 102=Let us C, 103=Operating System}
Values after remove: {101=Data Communication and Networking, 103=Operating System}
```

What is difference between HashMap and TreeMap?

HashMap	TreeMap
1) HashMap can contain one null key.	TreeMap can not contain any null key.

2) HashMap maintains no order.	TreeMap maintains ascending order.

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

- A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

Hashtable class declaration

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

1. **public class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

Hashtable class Parameters

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

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

Constructors of Java Hashtable class

Constructor	Description
Hashtable()	It is the default constructor of hash table it instantiates the Hashtable class.
Hashtable(int size)	It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size.
Hashtable(int size, float fillRatio)	It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.

Methods of Java Hashtable class

Method	Description
void clear()	It is used to reset the hash table.
boolean contains(Object value)	This method return true if some value equal to the value exist within the hash table, else return false.
boolean containsValue(Object value)	This method return true if some value equal to the value exists within the hash table, else return false.
boolean containsKey(Object key)	This method return true if some key equal to the key exists within the hash table, else return false.
boolean isEmpty()	This method return true if the hash table is empty; returns false if it contains at least one key.
void rehash()	It is used to increase the size of the hash table and rehashes all of its keys.
Object get(Object key)	This method return the object that contains the value associated with the key.
Object remove(Object key)	It is used to remove the key and its value. This method return the value associated with the key.
int size()	This method return the number of entries in the hash table.

Java Hashtable Example

```
import java.util.*;
class TestCollection16{
 public static void main(String args[]){
  Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

  hm.put(100,"Amit");
  hm.put(102,"Ravi");
```

```
        hm.put(101,"Vijay");
        hm.put(103,"Rahul");

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

Output:

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

Java Hashtable Example: remove()

```
import java.util.*;
public class HashtableExample {
  public static void main(String args[]) {
  // create and populate hash table
  Hashtable<Integer, String> map = new Hashtable<Integer, String>();
  map.put(102,"Let us C");
  map.put(103, "Operating System");
  map.put(101, "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: {103=Operating System, 102=Let us C, 101=Data Communication and Networking}
Values after remove: {103=Operating System, 101=Data Communication and Networking}
```

Difference between HashMap and Hashtable

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.
2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

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:

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

```
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 Comparator{
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);
}



}
}
```

```
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<Student>{
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
```

Properties class in Java

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 properties file and store data into properties file. Moreover, it can be used to get properties of system.

Advantage of properties file

Recompilation is not required, if information is changed from 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.

Methods of Properties class

The commonly used methods of Properties class are given below.

Method	Description
public void load(Reader r)	loads data from the Reader object.
public void load(InputStream is)	loads data from the InputStream object
public String getProperty(String key)	returns value based on the key.
public void setProperty(String key,String value)	sets the property in the properties object.
public void store(Writer w, String comment)	writers the properties in the writer object.
public void store(OutputStream os, String comment)	writes the properties in the OutputStream object.
storeToXML(OutputStream os, String comment)	writers the properties in the writer object for generating xml document.

public void storeToXML(Writer w, String comment, String encoding)	writers the properties in the writer object for generating xml document with specified encoding.

Example of Properties class to get information from properties file

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

db.properties

1. user=system
2. password=oracle

Now, lets 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 compile the java class again. That means no maintenance problem.

Example of Properties class to get all the system properties

By System.getProperties() method we can get all the properties of system. Let's create the class that gets information from the system properties.

Test.java

```
import java.util.*;
import java.io.*;
public class Test {
```

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

Properties p=System.getProperties();
Set set=p.entrySet();

Iterator itr=set.iterator();
while(itr.hasNext()){
Map.Entry entry=(Map.Entry)itr.next();
System.out.println(entry.getKey()+" = "+entry.getValue());
}

}
}
```

```
Output:
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = C:\Program Files\Java\jdk1.7.0_01\jre\bin
java.vm.version = 21.1-b02
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = ;
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
user.country = US
user.script =
sun.java.launcher = SUN_STANDARD
...........
```

Example of Properties class to create properties file

Now lets write the code to create the properties file.

Test.java

```
1. import java.util.*;
2. import java.io.*;
3. public class Test {
4. public static void main(String[] args)throws Exception{
5.
6. Properties p=new Properties();
7. p.setProperty("name","Sonoo Jaiswal");
8. p.setProperty("email","sonoojaiswal@javatpoint.com");
9.
10. p.store(new FileWriter("info.properties"),"Javatpoint Properties Example");
11.
12. }
13. }
```

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
```

Q26. Describe Arrays class in java.util package.

Answer :

Arrays is a class in java.util package introduced from JDK 1.5 version, which provides many static methods to do operations on arrays like filling, comparing, sorting and searching. Earlier to the introduction of Arrays, these operations were done from scratch. Now the coding has become easier with Arrays. Many static methods are overloaded.

Following is the class signature:

public class Arrays extends Object

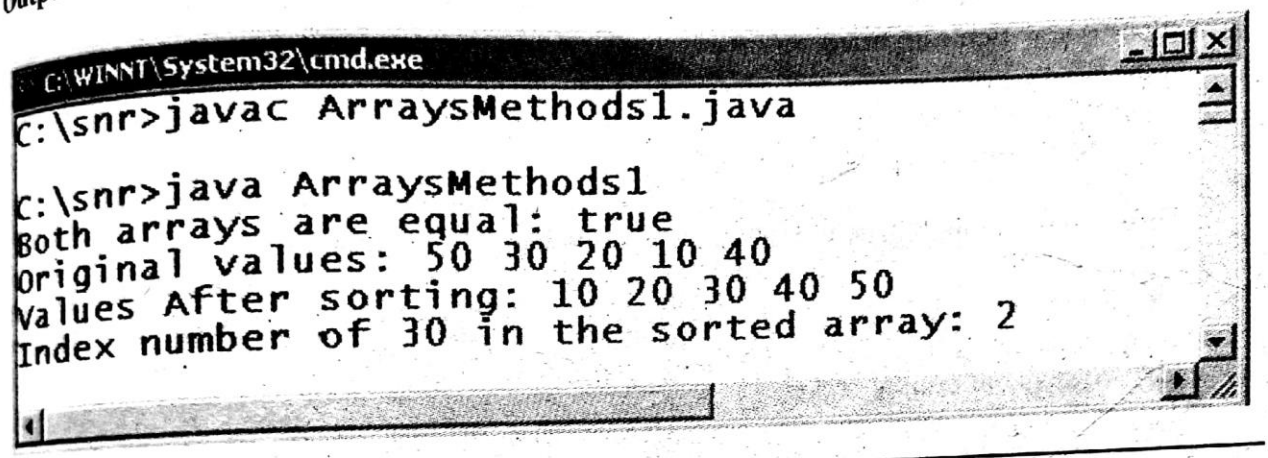
Following are some static methods with their description

1. **static int binarySearch(double myarray[], double num):** It is used to find a specified number (num) in the array (myarray) using binary search. This method is overloaded that can take a byte array, char array, int array etc.
2. **static void equals(double myarray1[], double myarray2[]):** It is used to check both the arrays have the same elements. This method is overloaded that can take a byte array, char array, int array etc.
3. **static void fill(double myarray1[], double num) :** It is used to fill all the array elements of the array (myarray1) with the specifeid value (num). This method is overloaded that can take a byte array, char array, int array etc.
4. **static void sort(double myarray1[]) :** It sorts the elemetns in the ascending order. This method is overloaded that can take a byte array, char array, int array etc.

```
import java.util.*;
public class ArraysMethods1
{
  public static void main(String[] args)
  {
     int marks[ ] = { 50,30, 20, 10, 40 };
     int price[ ] = { 50,30, 20, 10, 40 };
                                        // to compare two arrays whether their elements are same
     System.out.println("Both  arrays are equal: " + Arrays.equals(marks, price));
     System.out.print("Original values: " );
     for(int i = 0; i < marks.length; i++)
     {
          System.out.print(marks[i]+" ");
     }
      System.out.println( );
                                                                    // to give a new line
     Arrays.sort(marks);                    // sorting the elements of the array in ascending order
     System.out.print("Values After sorting: ");   // elements in the original array itself are changed
     for(int i = 0; i < marks.length; i++)
     {
          System.out.print(marks[i]+" ");
     }
```

```
                                        // to find the index number of an element
                                        // binarySearch( ) should be applied on a sorted array
    int k = Arrays.binarySearch(marks, 30);
    System.out.println("\nIndex number of 30 in the sorted array: "+k);
  }
}
```

Output:



Q27. What are legacy classes and interfaces?

Answer :

The data structures introduced with JDK 1.0 version are called legacy classes and interfaces. Even though, the Collections framework is far more superior with its rich set of features, the legacy classes are not deprecated. Instead, some more methods are added and reengineered to fit into collection framework.

Following gives a list of legacy classes and interfaces:

Legacy structure	Type of the class	Functionality
Stack	class(non-abstract)	Stores all types of objects(LIFO order)
Vector	class(non-abstract)	Stores all types of objects
Dictionary	abstract class	stores key/value pairs
Hashtable	class(non-abstract)	Stores key/value pairs
Properties	class(non-abstract)	Stores key/value pairs
Enumeration	interface	used to iterate and print the elements

Q28. Write a program using Stack.

Answer :

Stack is a well-known data structure with LIFO (Last-In-First-Out) pattern which means that the last item added to the stack is the first one to be removed.

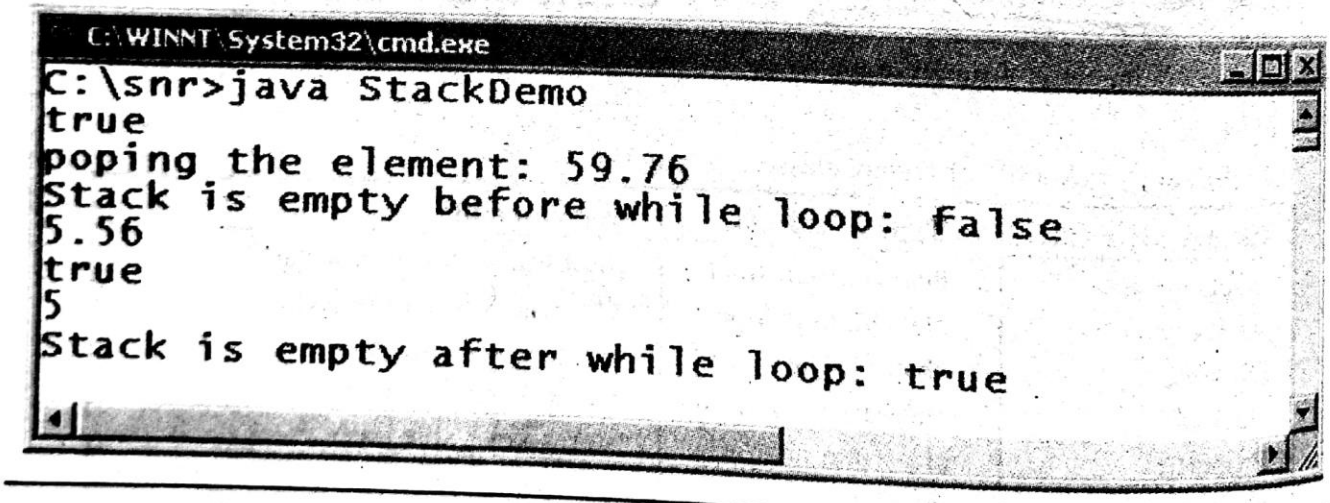
Following program illustrates stack operations like push( ), pop( ) etc.

File Name: StackDemo.java

```
import java.util.*;
public class StackDemo
{
   public static void main( String args[ ] )
   {
      Stack st = new Stack( ) ;
      System.out.println(st.empty( ));                                    // prints true
      st.push( new Integer( 5 ) ) ;                                       // empty( ) method returns true if
      st.push( new Boolean( true ) ) ;                                    // the stack is empty
      st.push( new Float( 5.56F ) ) ;
      st.push( new Double( 59.76 ) ) ;
      System.out.println( "poping the element: " + st.pop( ) ) ;
                                                                          // pops the last element added, that is 59.76
      System.out.println( "Stack is empty before while loop: " + st.empty( ) ) ;
                                                                          // returns false as stack is not empty
      while( ! st.empty( ) )                                              // iterates until all elements are popped out
      {
           System.out.println( st.pop( ) ) ;
      }

      System.out.println( "Stack is empty after while loop: " + st.empty( ) ) ;
   }                                                                      // prints true as all the elements are popped out
}
```

Output:



Q29. Illustrate Vector with an example? How it differs from ArrayList?

Answer :

Vector class is an expandable array of objects. It can grow at runtime as and when the elments are added to it, to accommodate the new elements. When elements are removed it shrinks automatically. It is like ArrayList, but with one differencee. Vector methods are synchronized where as ArrayList

methods are not synchronozied. For this reason, performance-wise ArrayList is better. With legacy vector, we use Enumeration interface instead of Iterator. But in J2SE 5, Vector is reengineered (re-modelled) to fit into Collections framework and now Vector can use generics and iterator.

Following is the class signature of Vector class (as in JDK 1.5):

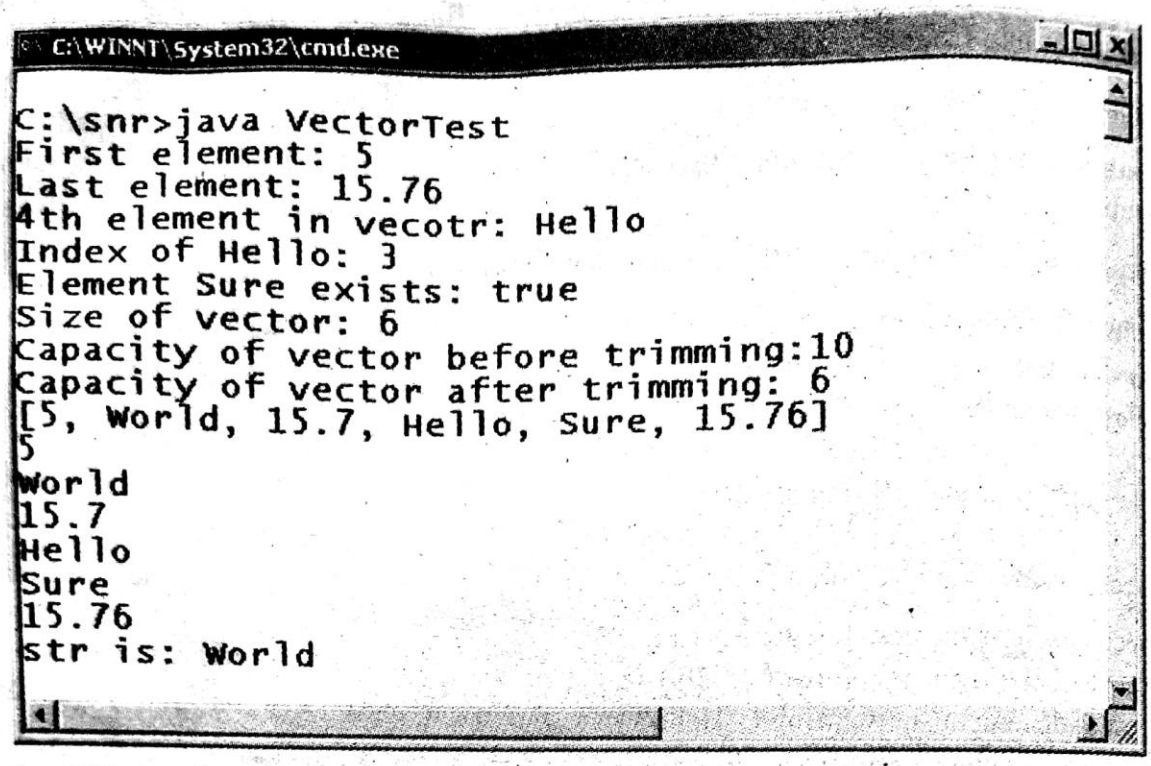
public class Vector extends AbstractList implements List, RandomAccess, Cloneable, Serializable

Following is the program that illustrates the methods of Vector.

File Name: VectorTest.java

```
import java.util.*;
public class VectorTest
{
  public static void main(String args[ ])
  {
    Vector vect = new Vector( ) ;                                    // default capacity 10
    vect.addElement( new Integer ( 5 ) ) ;
    vect.addElement( new Float ( 15.7F ) ) ;
    vect.addElement( new String ( "Hello" ) ) ;
    vect.addElement( "Sure" ) ;
    Double d = new Double( 15.76 ) ;
    vect.addElement( d ) ;
    String str = "World" ;
    vect.insertElementAt( str, 1 ) ;
    System.out.println( "First element: " + vect.firstElement( ) ) ;
    System.out.println( "Last element: " + vect.lastElement( ) ) ;
    System.out.println( "4th element in vecotr: " + vect.elementAt( 3 ) ) ;
    System.out.println( "Index of Hello: " + vect.indexOf( "Hello" ) ) ;
    System.out.println( "Element Sure exists: " + vect.contains( "Sure" ) ) ;
    System.out.println( "Size of vector: " + vect.size( ) ) ;                    //  prints 6
    System.out.println("Capacity of vector before trimming:"+vect.capacity( ));   //10
    vect.trimToSize( ) ;
    System.out.println( "Capacity of vector after trimming: " + vect.capacity( ));   // 6
    System.out.println( vect ) ;                                     // prints all elements in a line
    Enumeration e = vect.elements( ) ;          // assign the elements to Enumeration to print
    while( e.hasMoreElements( ) )
     {
          System.out.println( e.nextElement( ) ) ;
     }
    String s = (String) vect.elementAt( 1 ) ;
    System.out.println("str is: " + s ) ;
  }
}
```

Output:



Methods of Vector class used in the above program:

1. **new Vector( ):** constructs a Vector object of default capacity of 10
2. **void addElement(Object obj):** adds an element to the Vector
3. **void insertElementAt(Object obj, int index):** inserts the obj at the specified index. The first element gets 0 by default.
4. **Object firstElement( ):** returns the first element stored in the Vector
5. **Object lastElement( ):** returns the last element stored in the Vector
6. **Object elementAt(int index):** returns the element at the specified index
7. **int indexOf(Object obj):** returns the index number of the specifed element in the Vector
8. **boolean contains(Object obj):** returns true if the specified element exists
9. **int size( ):** returns the number of elements present in the Vector
9. **int capacity( ):** returns the storing capacity of the Vector
10. **void trimToSize( ):** deletes the extra capacity
11. **Enumeration elements( ):** returns an object of Enumeration interface

Q30. Discuss the Enumeration interface.

Answer :

The Enumeration interface provides a standard means of iterating through a list of sequentially stored elements in a data structure(this interface must be implemented by the data structure). It is replaced with Iterator interface in Collections framework.

It provides two abstract methods using which iteration can be done.

hasMoreElements( ) : returns false when no elements exists

nextElement( ) : retrieves the next element in the enumeration.

Method signatures of the above methods defined in the java.util package:

```
public abstract boolean hasMoreElements( ) ;
public abstract java.lang.Object nextElement( ) ;
```

Following snippet of code illustrates how iteration can be done.

(for full program, see the previous question)

```
Enumeration e = vect.elements( ) ;
while( e.hasMoreElements( ) )
{
    System.out.println( e.nextElement( ) ) ;
}
```

elements( ) method of Vector class returns an object of Enumeration interface. **hasMoreElements( )** returns true as long as elements in the Enumaration object. If all elements are exhausted to it returns false and the loop terminates. **nextElement( )** returns each object stored in the Enumeration interface object.

Q33. What is a StringTokenizer and illustrate with an example?

Answer :

The class java.util.StringTokenizer is a simpler version of a class StreamTokenizer (of java.io package), but it works only with strings. The set of delimiters (the characters that separate tokens) may be specified at creation time.

Sometimes, it is necessary to decompose a full line of data into tokens as required by a processor and interpreter etc. Each token is a word. We specify the delimiters based on which the string is to be tokenized. The default delimiter is whitespace like " \r\t\n" which will be taken by default we when omit any delimiter. Apart from this, we can specify our own group of delimiters.

Following is the class signature:

public class StringTokenizer extends Object implements Enumeration

Some important methods:

int countTokens( ): returns number of tokens.

boolean hasMoreTokens( ): returns true as long as tokens exist to return. If all tokens are exhausted, it returns false. It is used in while loop to print all the tokens.

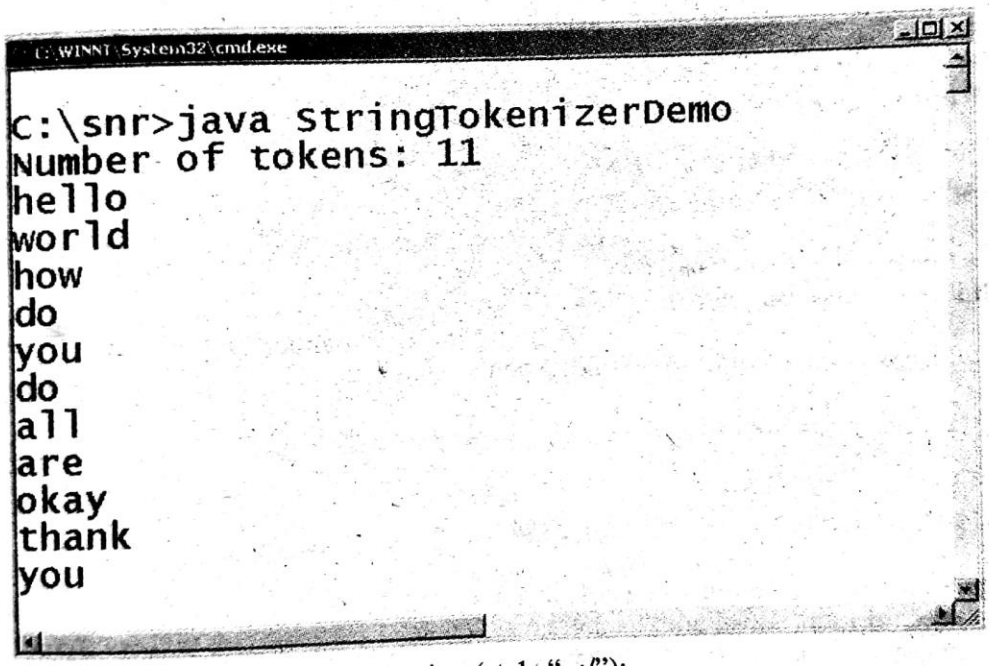
String nextToken( ): returns the next token available to return.

The following program illustrates a string tokenizer that takes whitespace and a group of our own delimiters.

File Name: StringTokezierDemo.java

```
import java.util.*;
public class StringTokenizerDemo
{
   public static void main(String args[])
   {
      String str1 = "hello world;how,do you/do,all are/okay;thank/you";
      StringTokenizer st = new StringTokenizer(str1, " ,;/");
      System.out.println("Number of tokens: " + st.countTokens( ) );
      while(st.hasMoreTokens( ) )
      {
          System.out.println(st.nextToken( ));
      }
   }
}
```

Output:



StringTokenizer st = new StringTokenizer(str1, " ,;/");

The first parameter of the StringTokenizer constructor takes the string to be parsed and the second parameter is the list of delimiters. Observe the whitespace before comma. The list of delimiters used are whitespace, comma, semicolon and forward slash.

Q34. Write short notes on BitSet.

Answer :

This class implements a vector of bits that grows as needed. Each component of the bit set has a boolean value. The bits of a BitSet are indexed by nonnegative integers. All the indexed bits can be examined, set, or cleared. One BitSet may be used to modify the contents of another BitSet through logical AND, logical inclusive OR, and logical exclusive OR operations.

Every bit set has a current size, which is the number of bits of space currently in use by the bit set. The class is useful for all bitset operations. By default, all bits in the set initially have the value false.

Following are some methods of BitSet class

void and(BitSet set)	:	performs a logical AND the invoking bit set with the argument bit set.
void andNot(BitSet set)	:	clears all of the bits in the invoking BitSet whose corresponding bit is set in the specified BitSet
void clear( )	:	sets all of the bits in this BitSet to false.
boolean get(int bitIndex)	:	returns the value of the bit with the specified index.
boolean isEmpty( )	:	returns true if this BitSet contains no bits that are set to true.
void or(BitSet set)	:	performs a logical OR of the invoking bit set with the bit set argument.
void xor(BitSet set)	:	performs a logical XOR of the invoking bit set with the bit set argument.

Following is the class signature:

public class BitSet extends Object implements Cloneable, Serializable

Following program illustrates the usage of BitSet.

File Name: BitSetInfo.java

```
import java.util.*;
public class BitSetOperations
{
    public static void main(String args[])
    {
                                        // creating BitSet objects
        BitSet bs1 = new BitSet(16);
        BitSet bs2 = new BitSet(16);
                                        // BitSet grows dynamically
        for(int i = 0; i < 16; i++)    // assigning some values
        {
            if( i % 2  == 0 )
                bs1.set( i );
            if( i % 5  != 0 )
                bs2.set( i );
        }
        System.out.println("Initial values of bit sets:");
        System.out.println("\tbs1 are: " + bs1);
        System.out.println("\tbs2 are: " + bs2);
        System.out.println("\nResults after operation\n");
                                        // using AND operation
        bs2.and(bs1);
        System.out.println("\tbs2 AND bs1 operation: " + bs2);
                                        // using OR operation
        bs2.or(bs1);
        System.out.println("\tbs2 OR bs1  operation: " + bs2);
                                        // using XOR operation
        bs2.xor(bs1);
        System.out.println("\tbs2 XOR bs1 operation: " + bs2);
    }
}
```

Output:

```
C:\WINNT\System32\cmd.exe
C:\snr>javac BitSetOperations.java

C:\snr>java BitSetOperations
Initial values of bit sets:
        bs1 are: {0, 2, 4, 6, 8, 10, 12, 14}
        bs2 are: {1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

Results after operation

        bs2 AND bs1 operation: {2, 4, 6, 8, 12, 14}
        bs2 OR bs1  operation: {0, 2, 4, 6, 8, 10, 12, 14}
        bs2 XOR bs1 operation: {}
```

Q35. Write notes on Date class with an example.

OR

Write a program using Date class to print system time in hrs, mts and secs.

Answer :

The Date class represents system's current date including time particulars. JDK 1.1 introduced Calendar and DateFormat classes with more date manipulations. For this reason, many methods of Date class are deprecated (in favor of Calendar class). Java treats 1st January, 1970 as standard date called as Epoch date.

Following is the class signature of Date class:

public class Date extends Object implements Serializable, Cloneable, Comparable

The class `Date` represents a specific instant in time, with millisecond precision.

The Date class constructor is overloaded six times of which some important are given

1. **Date( ):** It crates a Date object with current sytem time. This object gives complete informaiton of the date like year, month, date, hours, minutes, seconds and milliseconds etc.
2. **Date(int y, int m, int d):** It creates Data object with year, month and date. It does not give other information.

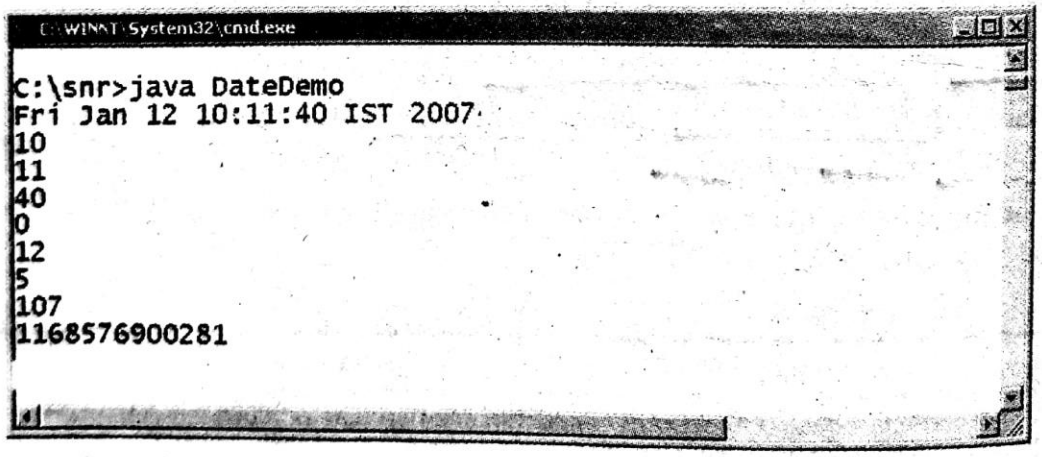
Some important methos

Method	Descritpion
1. boolean equals(Object obj) :	compares two date objects.
2. int getMinutes( ) :	returns the minutes part of the currrent time.
3. int getHours( ) :	returns the hours part of the currrent time.
4. int getDate( ) :	returns the date part of the currrent time.
5. int getYear( ) :	returns the year part of the currrent time (it givescurrent_year-1900).

File Name: DateDemo.java

```
import java.util.Date;
public class DateDemo
{
   public static void main(String args[])
   {
      Date d = new Date( );                                  // Thu Jan 01 07.30.33 GMT+05.30 2004
      System.out.println(d);
                                                             // to extract hours
      System.out.println(d.getHours( ));                     // prints 10
                                                             // to extract minutes
      System.out.println(d.getMinutes( ));                   // prints 11
                                                             // to extract seconds
      System.out.println(d.getSeconds( ));                   // prints 40
                                                             // to extract hours
      System.out.println(d.getMonth( ));                     // prints 0 (for Jan)
                                                             // to extract date
      System.out.println(d.getDate( ));                      // prints 12
                                                             // to extract day
      System.out.println(d.getDay( ));                       // prints 5 (for Friday)
                                                             // to extract year
      System.out.println(d.getYear());                       // prints 107 (add 1900, the epoch year))
                                                             // to extract time
      System.out.println(d.getTime( ));
                // prints 1168576900281. Prints milliseconds from Jan 1, 1900(called epoch time)
      }
}
```

Output:



Q36. Write about Calendar class?

Answer :

The Calendar class is an abstract class that provides methods for converting date and time to a set of calendar fields such as HOUR, MONTH, MINUTE etc. We can also manipulate the calendar

fields to get the date in a coming week or to know any year is before after a specified year. An object of time can be represented by a millisecond value that is from the Epoch, January 1, 1970 mid-night GMT (Gregorian).

Following is the signature of Calendar class:

public abstract class Calendar extends Object implements **Serializable, Cloneable, Comparable**

Following program illustrate the usage of Calendar class variables and methods:

File Name: UsingCalendar.java

```
import java.util.*;
public class UsingCalendar
{
  public static void main(String args[])
  {
     Calendar cal = Calendar.getInstance( );
                                                    // getting system date particulars
     System.out.println("Date : " + cal.get(Calendar.DATE));
     System.out.println("Month : " + cal.get(Calendar.MONTH));          // prints 0 for Jan
     System.out.println("Year : " + cal.get(Calendar.YEAR));
  }
}
```

Output: