Unit 4

Collections Introduction

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

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.).

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.

Need for Collection Framework :

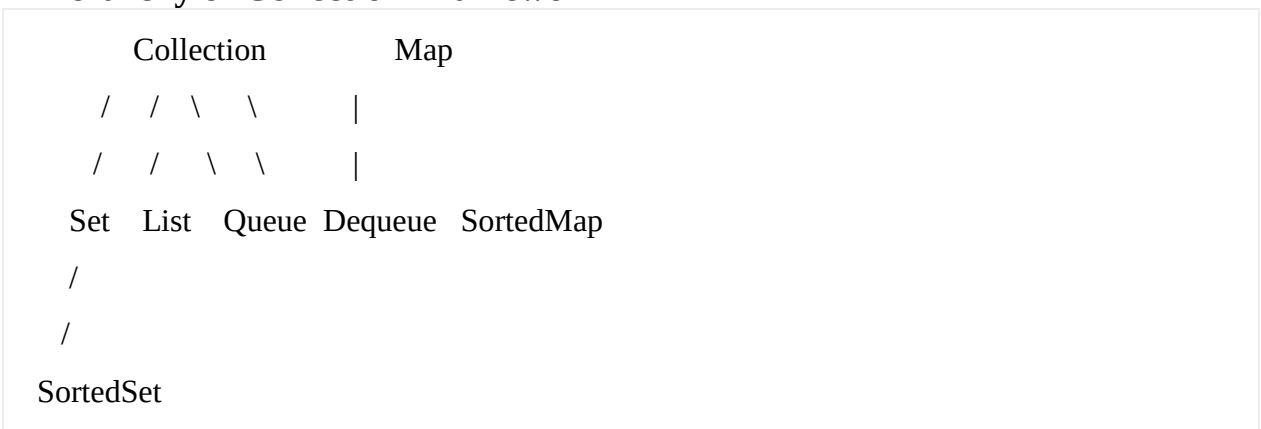
Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members.

Advantages of Collection Framework:

1. **Consistent API** : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some*common set of methods.
2. **Reduces programming effort:** A programmer doesn't have to worry about the design of Collection, and he can focus on its best use in his program.
3. **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms.

Hierarchy of Collection Framework



Core Interfaces in Collections

Note that this diagram only shows core interfaces.
Collection : Root interface with basic methods like add(), remove(), contains(), isEmpty(), addAll(), ... etc.

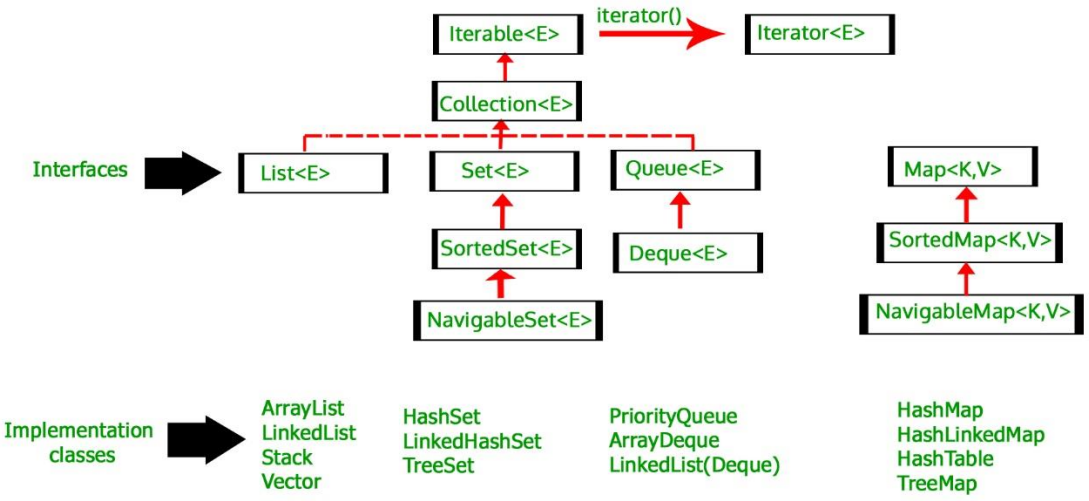
Set : Doesn't allow duplicates. Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based). Note that TreeSet implements **SortedSet**.

List : Can contain duplicates and elements are ordered. Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

Queue : Typically order elements in FIFO order except exceptions like PriorityQueue.
Deque : Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.

Map : Contains Key value pairs. Doesn't allow duplicates. Example implementation are HashMap and TreeMap.
TreeMap implements **SortedMap**.

The difference between Set and Map interface is that in Set we have only keys, whereas in Map, we have key, value pairs.



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.
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.

Iterator interface

Iterator interface provides the facility of iterating the elements in a 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 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.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

which returns the iterator over the elements of type T.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have the duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in ArrayList class can be randomly accessed. Consider the following example.

```
1. import java.util.*;
2. class TestJavaCollection1{
3. public static void main(String args[]){
4. ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection2{
3. public static void main(String args[]){
4. LinkedList<String> al=new LinkedList<String>();
```

```
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3. public static void main(String args[]){
4. Vector<String> v=new Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its own methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection4{
3. public static void main(String args[]){
4. Stack<String> stack = new Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, ArrayDeque, etc. which implements the Queue interface.

Queue interface can be instantiated as:

```
1. Queue<String> q1 = new PriorityQueue();
2. Queue<String> q2 = new ArrayDeque();
```

There are various classes that implements the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection5{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
1. Deque d = new ArrayDeque();
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection6{
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. for (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

```
1. Set<data-type> s1 = new HashSet<data-type>();
2. Set<data-type> s2 = new LinkedHashSet<data-type>();
3. Set<data-type> s3 = new TreeSet<data-type>();
```

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains the unique items.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection7{
3. public static void main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=new HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection8{
```

```
3. public static void main(String args[]){
4. LinkedHashSet<String> set=new LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
1. SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains the unique elements However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

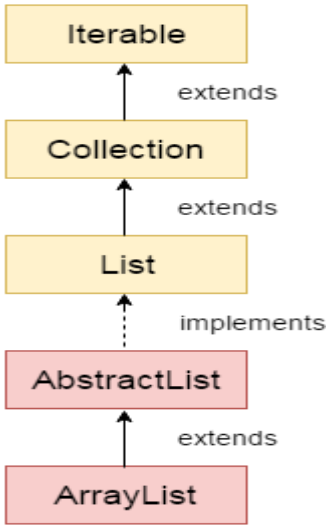
```
1. import java.util.*;
2. public class TestJavaCollection9{
3. public static void main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=new TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
```

```
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

Ajay
Ravi
Vijay

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, 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.
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*.

Java ArrayList Example

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

```
2. class TestCollection1{
3.  public static void main(String args[]){
4.   ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.   list.add("Ravi");//Adding object in arraylist
6.   list.add("Vijay");
7.   list.add("Ravi");
8.   list.add("Ajay");
9.   //Traversing list through Iterator
10.  Iterator itr=list.iterator();
11.  while(itr.hasNext()){
12.   System.out.println(itr.next());
13.  }
14. }
15. }
```

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

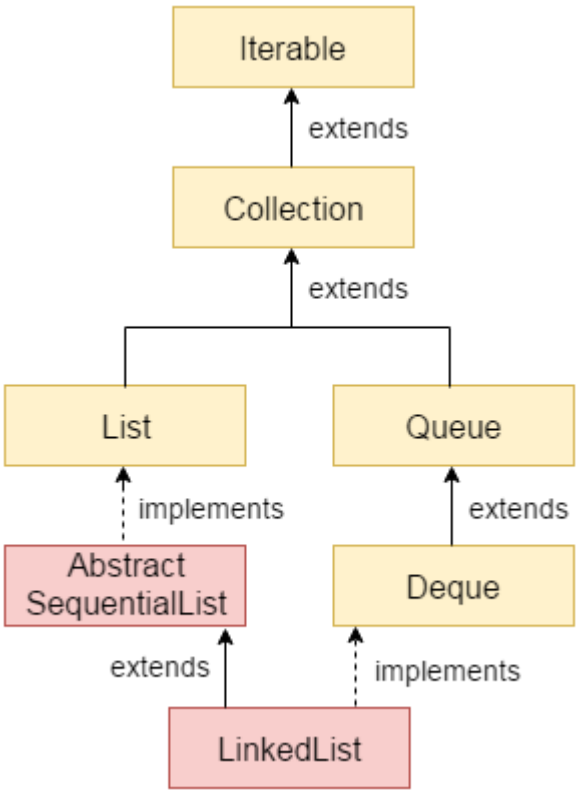
```
1. import java.util.*;
2. class TestCollection2{
3.  public static void main(String args[]){
4.   ArrayList<String> al=new ArrayList<String>();
```

```
5.  al.add("Ravi");
6.  al.add("Vijay");
7.  al.add("Ravi");
8.  al.add("Ajay");
9.  for(String obj:al)
10.   System.out.println(obj);
11. }
12. }
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

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>, 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.
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

```
1. import java.util.*;
2. public class TestCollection7{
3.  public static void main(String args[]){
4.
5.   LinkedList<String> al=new LinkedList<String>();
6.   al.add("Ravi");
7.   al.add("Vijay");
8.   al.add("Ravi");
9.   al.add("Ajay");
10.
11.  Iterator<String> itr=al.iterator();
12.  while(itr.hasNext()){
13.   System.out.println(itr.next());
14.  }
15. }
16. }
```

OUTPUT

Output:Ravi
Vijay
Ravi
Ajay

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	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.

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.

Example of ArrayList and LinkedList in Java

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

```
1. import java.util.*;
2. class TestArrayLinked{
3.  public static void main(String args[]){
4.
5.   List<String> al=new ArrayList<String>();//creating arraylist
6.   al.add("Ravi");//adding object in arraylist
7.   al.add("Vijay");
8.   al.add("Ravi");
9.   al.add("Ajay");
10.
11.  List<String> al2=new LinkedList<String>();//creating linkedlist
12.  al2.add("James");//adding object in linkedlist
13.  al2.add("Serena");
14.  al2.add("Swati");
15.  al2.add("Junaid");
16.
17.  System.out.println("arraylist: "+al);
18.  System.out.println("linkedlist: "+al2);
19. }
20. }
```

Output:

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

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>
```

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

```
1. import java.util.*;
2. public class ListExample{
3. public static void main(String args[]){
```

```
4.  ArrayList<String> al=new ArrayList<String>();
5.  al.add("Amit");
6.  al.add("Vijay");
7.  al.add("Kumar");
8.  al.add(1,"Sachin");
9.  System.out.println("Element at 2nd position: "+al.get(2));
10. for(String s:al){
11.  System.out.println(s);
12. }
13. }
14. }
```

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.

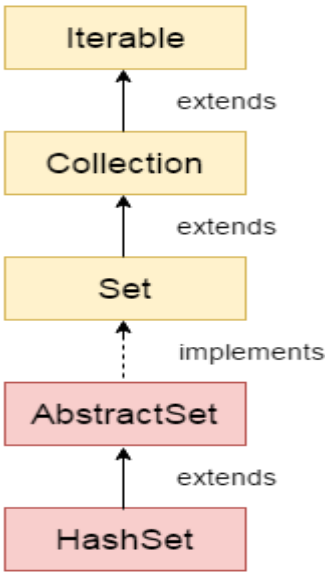
Example of ListIterator Interface

```
1. import java.util.*;
2. public class TestCollection8{
3. public static void main(String args[]){
4. ArrayList<String> al=new ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. System.out.println("element at 2nd position: "+al.get(2));
10. ListIterator<String> itr=al.listIterator();
11. System.out.println("traversing elements in forward direction...");
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. System.out.println("traversing elements in backward direction...");
16. while(itr.hasPrevious()){
17. System.out.println(itr.previous());
18. }
19. }
20. }
```

Output:

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

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

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

Various methods of Java HashSet class are as follows:

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.
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.

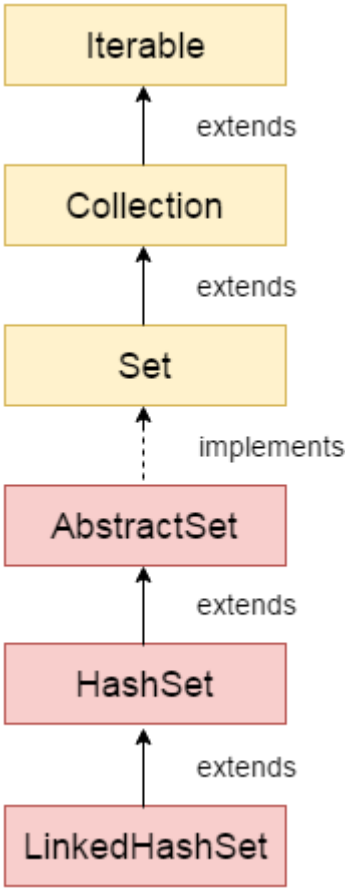
Java HashSet Example

```
1. import java.util.*;
2. class TestCollection9{
3.  public static void main(String args[]){
4.   //Creating HashSet and adding elements
5.   HashSet<String> set=new HashSet<String>();
6.   set.add("Ravi");
7.   set.add("Vijay");
8.   set.add("Ravi");
9.   set.add("Ajay");
10.  //Traversing elements
11.  Iterator<String> itr=set.iterator();
12.  while(itr.hasNext()){
13.   System.out.println(itr.next());
14.  }
15. }
16. }
```

Output

```
Ajay
Vijay
Ravi
```

Java LinkedHashSet class



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

The important points about Java LinkedHashSet class are:

- Contains unique elements only like HashSet.
- Provides all optional set operations, and permits null elements.
- Maintains insertion order.

Hierarchy of LinkedHashSet class

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

LinkedHashSet class declaration

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

1. **public class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Seria lizable

Constructors of Java LinkedHashSet 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.
LinkedHashSet(int capacity)	It is used initialize the capacity of the linkedhashset to the given integer value capacity.
LinkedHashSet(int capacity, float fillRatio)	It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument.

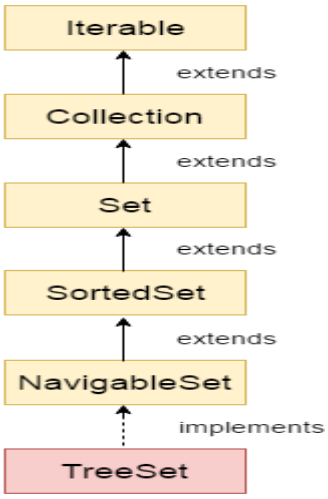
Example of LinkedHashSet class:

```
1. import java.util.*;
2. class TestCollection10{
3.  public static void main(String args[]){
4.   LinkedHashSet<String> al=new LinkedHashSet<String>();
5.   al.add("Ravi");
6.   al.add("Vijay");
7.   al.add("Ravi");
8.   al.add("Ajay");
9.   Iterator<String> itr=al.iterator();
10.  while(itr.hasNext()){
11.   System.out.println(itr.next());
12.  }
13. }
14. }
```

Output

```
Ravi
Vijay
Ajay
```

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, 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.
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

```
1. import java.util.*;
2. class TestCollection11{
3.  public static void main(String args[]){
4.   //Creating and adding elements
5.   TreeSet<String> al=new TreeSet<String>();
6.   al.add("Ravi");
7.   al.add("Vijay");
8.   al.add("Ravi");
9.   al.add("Ajay");
10.  //Traversing elements
11.  Iterator<String> itr=al.iterator();
12.  while(itr.hasNext()){
13.   System.out.println(itr.next());
14.  }
15. }
16. }
```

Output:

```
Ajay
Ravi
Vijay
```

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

```
1. import java.util.*;
2. class TestCollection12{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit");
6. queue.add("Vijay");
7. queue.add("Karan");
8. queue.add("Jai");
```

```
9.  queue.add("Rahul");
10. System.out.println("head:"+queue.element());
11. System.out.println("head:"+queue.peek());
12. System.out.println("iterating the queue elements:");
13. Iterator itr=queue.iterator();
14. while(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. while(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }
```

Output:

```
        head:Amit
    head:Amit
    iterating the queue elements:
    Amit
    Jai
    Karan
    Vijay
    Rahul
    after removing two elements:
    Karan
    Rahul
    Vijay
```

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".**

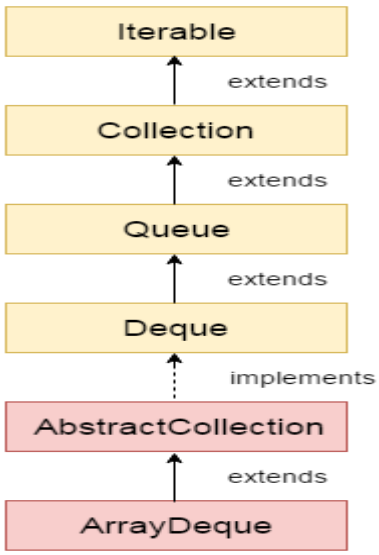
Deque Interface declaration

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>, Clone
   able, Serializable
```

Java ArrayDeque Example

```
1. import java.util.*;
2. public class ArrayDequeExample {
3.    public static void main(String[] args) {
4.    //Creating Deque and adding elements
5.    Deque<String> deque = new ArrayDeque<String>();
6.    deque.add("Ravi");
7.    deque.add("Vijay");
8.    deque.add("Ajay");
9.    //Traversing elements
10.   for (String str : deque) {
11.   System.out.println(str);
12.   }
13.   }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

Java ArrayDeque Example: offerFirst() and pollLast()

```
1. import java.util.*;
2. public class DequeExample {
3. public static void main(String[] args) {
4.     Deque<String> deque=new ArrayDeque<String>();
5.     deque.offer("arvind");
```

```
6.      deque.offer("vimal");
7.      deque.add("mukul");
8.      deque.offerFirst("jai");
9.      System.out.println("After offerFirst Traversal...");
10.     for(String s:deque){
11.         System.out.println(s);
12.     }
13.     //deque.poll();
14.     //deque.pollFirst();//it is same as poll()
15.     deque.pollLast();
16.     System.out.println("After pollLast() Traversal...");
17.     for(String s:deque){
18.         System.out.println(s);
19.     }
20. }
21. }
```

Output:

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

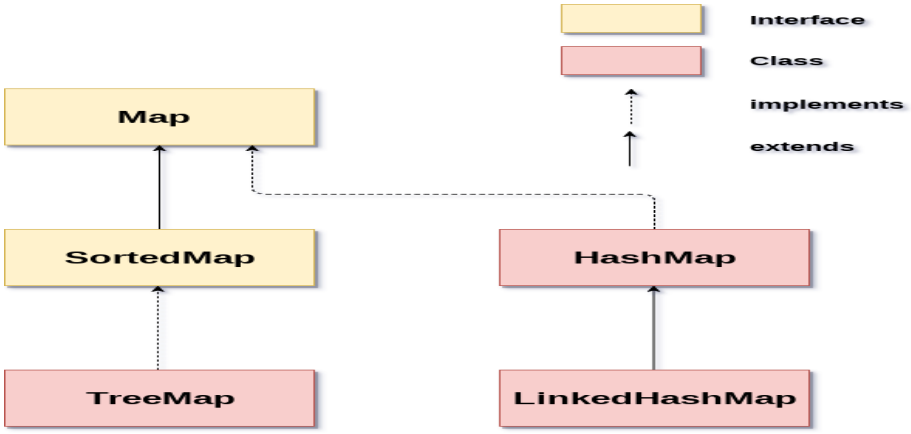
Java Map Interface

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:



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)

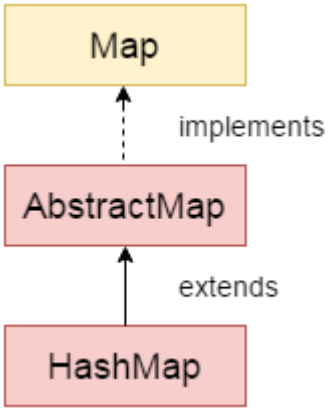
```
1. import java.util.*;
2. class MapInterfaceExample{
3.  public static void main(String args[]){
4.   Map<Integer,String> map=new HashMap<Integer,String>();
5.   map.put(100,"Amit");
6.   map.put(101,"Vijay");
7.   map.put(102,"Rahul");
8.   for(Map.Entry m:map.entrySet()){
```

```
9.      System.out.println(m.getKey()+" "+m.getValue());
10.   }
11.  }
12. }
```

Output:

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

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, 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.
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

```
1. import java.util.*;
2. class TestCollection13{
3.  public static void main(String args[]){
4.   HashMap<Integer,String> hm=new HashMap<Integer,String>();
5.   hm.put(100,"Amit");
6.   hm.put(101,"Vijay");
7.   hm.put(102,"Rahul");
8.   for(Map.Entry m:hm.entrySet()){
9.    System.out.println(m.getKey()+" "+m.getValue());
10.  }
11. }
12. }
```

Output:102 Rahul
100 Amit
101 Vijay

Java HashMap Example: remove()

```
1. import java.util.*;
2. public class HashMapExample {
```

```
3.    public static void main(String args[]) {
4.    // create and populate hash map
5.    HashMap<Integer, String> map = new HashMap<Integer, String>();
6.    map.put(101,"Let us C");
7.    map.put(102, "Operating System");
8.    map.put(103, "Data Communication and Networking");
9.    System.out.println("Values before remove: "+ map);
10.   // Remove value for key 102
11.   map.remove(102);
12.   System.out.println("Values after remove: "+ map);
13.   }
14. }
```

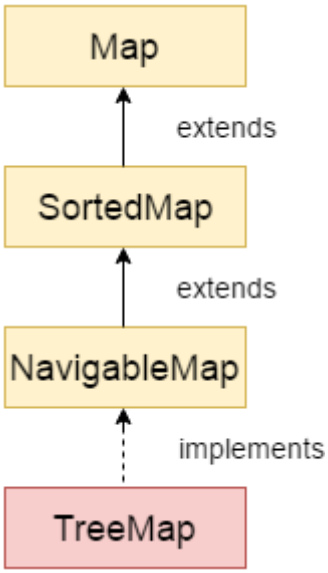
Output:

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 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, Serializable

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:

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

```
4.    TreeMap<Integer,String> hm=new TreeMap<Integer,String>();
5.    hm.put(100,"Amit");
6.    hm.put(102,"Ravi");
7.    hm.put(101,"Vijay");
8.    hm.put(103,"Rahul");
9.    for(Map.Entry m:hm.entrySet()){
10.    System.out.println(m.getKey()+" "+m.getValue());
11.   }
12.  }
13. }
```

Output:100 Amit
101 Vijay
102 Ravi
103 Rahul

Java TreeMap Example: remove()

```
1.  import java.util.*;
2.  public class TreeMapExample {
3.     public static void main(String args[]) {
4.     // Create and populate tree map
5.     Map<Integer, String> map = new TreeMap<Integer, String>();
6.     map.put(102,"Let us C");
7.     map.put(103, "Operating System");
8.     map.put(101, "Data Communication and Networking");
9.     System.out.println("Values before remove: "+ map);
10.    // Remove value for key 102
11.    map.remove(102);
12.    System.out.println("Values after remove: "+ map);
13.    }
14. }
```

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

```
1. import java.util.*;
2. class TestCollection16{
3.  public static void main(String args[]){
4.   Hashtable<Integer,String> hm=new Hashtable<Integer,String>();
5.
6.   hm.put(100,"Amit");
7.   hm.put(102,"Ravi");
8.   hm.put(101,"Vijay");
9.   hm.put(103,"Rahul");
10.
11.  for(Map.Entry m:hm.entrySet()){
12.   System.out.println(m.getKey()+" "+m.getValue());
13.  }
14. }
15. }
```

Output:

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

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 (Generic)

Student.java

```
1. class Student{
2. int rollno;
3. String name;
4. int age;
5. Student(int rollno,String name,int age){
6. this.rollno=rollno;
7. this.name=name;
8. this.age=age;
9. }
10. }
```

AgeComparator.java

```
1. import java.util.*;
2. class AgeComparator implements Comparator<Student>{
3. public int compare(Student s1,Student s2){
4. if(s1.age==s2.age)
5. return 0;
6. else if(s1.age>s2.age)
7. return 1;
8. else
9. return -1;
10. }
11. }
```

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.

```
1. import java.util.*;
2. class NameComparator implements Comparator<Student>{
3. public int compare(Student s1,Student s2){
4. return s1.name.compareTo(s2.name);
5. }
6. }
```

Simple.java

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

```
1. import java.util.*;
2. import java.io.*;
```

```
3. class Simple{
4. public static void main(String args[]){
5.
6. ArrayList<Student> al=new ArrayList<Student>();
7. al.add(new Student(101,"Vijay",23));
8. al.add(new Student(106,"Ajay",27));
9. al.add(new Student(105,"Jai",21));
10.
11. System.out.println("Sorting by Name...");
12.
13. Collections.sort(al,new NameComparator());
14. for(Student st: al){
15. System.out.println(st.rollno+" "+st.name+" "+st.age);
16. }
17.
18. System.out.println("sorting by age...");
19.
20. Collections.sort(al,new AgeComparator());
21. for(Student st: al){
22. System.out.println(st.rollno+" "+st.name+" "+st.age);
23. }
24.
25. }
26. }
```

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

```
1. import java.util.*;
2. import java.io.*;
3. public class Test {
4. public static void main(String[] args)throws Exception{
5.     FileReader reader=new FileReader("db.properties");
6.
7.     Properties p=new Properties();
8.     p.load(reader);
9.
10.    System.out.println(p.getProperty("user"));
11.    System.out.println(p.getProperty("password"));
12. }
13. }
```

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

```
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=System.getProperties();
7. Set set=p.entrySet();
8.
9. Iterator itr=set.iterator();
10. while(itr.hasNext()){
11. Map.Entry entry=(Map.Entry)itr.next();
12. System.out.println(entry.getKey()+" = "+entry.getValue());
13. }
```

14.
15. }
16. }

```
Output:
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = C:\Program Files\Java\jdk1.8.0_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
.....
```

Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

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

ArrayList	**Vector**
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.

5) ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

Legacy Classes

Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects. When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes. All legacy classes and interface were redesign by JDK 5 to support Generics. In general, the legacy classes are supported because there is still some code that uses them.

The following are the legacy classes defined by **java.util** package

1. Dictionary
2. HashTable
3. Properties
4. Stack
5. Vector

There is only one legacy interface called **Enumeration**

NOTE: All the legacy classes are synchronized

Enumeration interface

1. **Enumeration** interface defines method to enumerate(obtain one at a time) through collection of objects.
2. This interface is superseded(replaced) by **Iterator** interface.
3. However, some legacy classes such as **Vector** and **Properties** defines several method in which **Enumeration** interface is used.
4. It specifies the following two methods

   boolean hasMoreElements() //It returns true while there are still more elements to extract, and returns false when all the elements have been enumerated.

   Object nextElement() //It returns the next object in the enumeration i.e. each call to nextElement() method

obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.

Vector class

1. **Vector** is similar to **ArrayList** which represents a dynamic array.
2. There are two differences between **Vector** and **ArrayList**. First, Vector is synchronized while ArrayList is not, and Second, it contains many legacy methods that are not part of the Collections Framework.
3. With the release of JDK 5, Vector also implements Iterable. This means that Vector is fully compatible with collections, and a Vector can have its contents iterated by the for-each loop.
4. Vector class has following four constructor
5. Vector() //This creates a default vector, which has an initial size of 10.
6.
7. Vector(int size) //This creates a vector whose initial capacity is specified by size.
8.
9. Vector(int size, int incr) //This creates a vector whose initial capacity is specified by size and whose
10. increment is specified by incr. The increment specifies the number of elements to allocate each time
11. when a vector is resized for addition of objects.
12. Vector(Collection c) //This creates a vector that contains the elements of collection c.

Vector defines several legacy methods. Lets see some important legacy methods defined by **Vector**class.

Method	**Description**
void addElement(E element)	adds element to the Vector
E elementAt(int index)	returns the element at specified index
Enumeration elements()	returns an enumeration of element in vector

E firstElement()	returns first element in the Vector
E lastElement()	returns last element in the Vector
void removeAllElements()	removes all elements of the Vector

Example of Vector

```
import java.util.*;
public class Test
{
  public static void main(String[] args)
   {
      Vector ve = new Vector();
       ve.add(10);
       ve.add(20);
       ve.add(30);
       ve.add(40);
       ve.add(50);
       ve.add(60);

       Enumeration en = ve.elements();

       while(en.hasMoreElements())
       {
          System.out.println(en.nextElement());
       }
   }

}
```

```
10
20
30
```

40

50

60

Hashtable class

1. Like HashMap, Hashtable also stores key/value pair. However

   neither **keys** nor **values** can be **null**.
2. There is one more difference between **HashMap** and **Hashtable** that is Hashtable is

   synchronized while HashMap is not.
3. Hashtable has following four constructor
4. Hashtable() //This is the default constructor. The default size is 11.
5.
6. Hashtable(int size) //This creates a hash table that has an initial size specified by size.
7.
8. Hashtable(int size, float fillratio) //This creates a hash table that has an initial size specified by size
9. and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full
10. the hash table can be before it is resized upward. Specifically, when the number of elements is greater
11. than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded.
12. If you do not specify a fill ratio, then 0.75 is used.
13. Hashtable(Map< ? extends K, ? extends V> m) //This creates a hash table that is initialized with the
14. elements in m. The capacity of the hash table is set to twice the number of elements in m.

    The default load factor of 0.75 is used.

Example of Hashtable

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

```
  ht.put("a",new Integer(100));
  ht.put("b",new Integer(200));
  ht.put("c",new Integer(300));
  ht.put("d",new Integer(400));

  Set st = ht.entrySet();
  Iterator itr=st.iterator();
  while(itr.hasNext())
  {
   Map.Entry m=(Map.Entry)itr.next();
   System.out.println(itr.getKey()+" "+itr.getValue());
  }
 }
}
```

a 100

b 200

c 300

d 400

Difference between HashMap and Hashtable

Hashtable	**HashMap**
Hashtable class is synchronized.	HashMap is not synchronized.
Because of Thread-safe, Hashtable is slower than HashMap	HashMap works faster.
Neither **key** nor **values** can be null	Both **key** and **values** can be null
Order of table remain constant over time.	does not guarantee that order of map will remain constant over time.

Properties class

1. **Properties** class extends **Hashtable** class.
2. It is used to maintain list of value in which both key and value are **String**
3. **Properties** class define two constructor
4. Properties() //This creates a Properties object that has no default values
5. Properties(Properties propdefault) //This creates an object that uses propdefault for its default values.
6. One advantage of **Properties** over **Hashtable** is that we can specify a default property that will be useful when no value is associated with a certain key.

   Note: In both cases, the property list is empty
7. In Properties class, you can specify a default property that will be returned if no value is associated with a certain key.

Example of Properties class

```
import java.util.*;

public class Test
{
  public static void main(String[] args)
  {
    Properties pr = new Properties();
    pr.put("Java", "James Ghosling");
    pr.put("C++", "Bjarne Stroustrup");
    pr.put("C", "Dennis Ritchie");
    pr.put("C#", "Microsoft Inc.");
    Set< ?> creator = pr.keySet();

    for(Object ob: creator)
    {
       System.out.println(ob+" was created by "+ pr.getProperty((String)ob) );
    }

  }
```

```
}
```

Java was created by James Ghosling

C++ was created by Bjarne Stroustrup

C was created by Dennis Ritchie

C# was created by Microsoft Inc

Stack class

1. Stack class extends Vector.
2. It follows last-in, first-out principle for the stack elements.
3. It defines only one default constructor

   Stack() //This creates an empty stack

4. If you want to put an object on the top of the stack, call push() method. If you want to remove and return the top element, call pop() method. An EmptyStackException is thrown if you call pop() method when the invoking stack is empty.

You can use peek() method to return, but not remove, the top object. The empty() method returns true if nothing is on the stack. The search() method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack.

Example of Stack

```
import java.util.*;


class StackDemo {
public static void main(String args[]) {
Stack st = new Stack();
st.push(11);
st.push(22);
st.push(33);
st.push(44);
st.push(55);
Enumeration e1 = st.elements();
```

```
while(e1.hasMoreElements())
System.out.print(e1.nextElement()+" ");

st.pop();
st.pop();

System.out.println("\nAfter popping out two elements");

Enumeration e2 = st.elements();

while(e2.hasMoreElements())
System.out.print(e2.nextElement()+" ");

}
}
```

```
11 22 33 44 55
After popping out two elements
11 22 33
```

Dictionary class

1. Dictionary is an abstract class.
2. It represents a key/value pair and operates much like Map.
3. Although it is not currently deprecated, Dictionary is classified as obsolete, because it is fully superseded by Map class.

StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimeter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of StringTokenizer class

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

Simple example of StringTokenizer class

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
1. import java.util.StringTokenizer;
2. public class Simple{
3.  public static void main(String args[]){
4.    StringTokenizer st = new StringTokenizer("my name is khan"," ");
5.      while (st.hasMoreTokens()) {
6.          System.out.println(st.nextToken());
7.      }
8.    }
9. }
```

Output:my
name
is
khan

Example of nextToken(String delim) method of StringTokenizer class

```
1. import java.util.*;
2.
3. public class Test {
4.    public static void main(String[] args) {
5.        StringTokenizer st = new StringTokenizer("my,name,is,khan");
6.
7.       // printing next token
8.       System.out.println("Next token is : " + st.nextToken(","));
9.    }
10. }
```

Output:Next token is : my

StringTokenizer class is deprecated now. It is recommended to use split() method of String class or regex (Regular Expression).

Java - The BitSet Class

The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

The BitSet defines the following two constructors.

Sr.No.	Constructor & Description
1	**BitSet( )** This constructor creates a default object.
2	**BitSet(int size)** This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero.

BitSet implements the Cloneable interface and defines the methods listed in the following table −

Sr.No.	Method & Description
1	**void and(BitSet bitSet)** ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.
2	**void andNot(BitSet bitSet)** For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	**int cardinality( )** Returns the number of set bits in the invoking object.
4	**void clear( )** Zeros all bits.
5	**void clear(int index)** Zeros the bit specified by index.
6	**void clear(int startIndex, int endIndex)** Zeros the bits from startIndex to endIndex.

7	**Object clone( )** Duplicates the invoking BitSet object.
8	**boolean equals(Object bitSet)** Returns true if the invoking bit set is equivalent to the one passed in bitSet. Otherwise, the method returns false.
9	**void flip(int index)** Reverses the bit specified by the index.
10	**void flip(int startIndex, int endIndex)** Reverses the bits from startIndex to endIndex.
11	**boolean get(int index)** Returns the current state of the bit at the specified index.
12	**BitSet get(int startIndex, int endIndex)** Returns a BitSet that consists of the bits from startIndex to endIndex. The invoking object is not changed.
13	**int hashCode( )** Returns the hash code for the invoking object.
14	**boolean intersects(BitSet bitSet)** Returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
15	**boolean isEmpty( )** Returns true if all bits in the invoking object are zero.
16	**int length( )** Returns the number of bits required to hold the contents of the invoking BitSet.

	This value is determined by the location of the last 1 bit.
17	**int nextClearBit(int startIndex)** Returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex.
18	**int nextSetBit(int startIndex)** Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned.
19	**void or(BitSet bitSet)** ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.
20	**void set(int index)** Sets the bit specified by index.
21	**void set(int index, boolean v)** Sets the bit specified by index to the value passed in v. True sets the bit, false clears the bit.
22	**void set(int startIndex, int endIndex)** Sets the bits from startIndex to endIndex.
23	**void set(int startIndex, int endIndex, boolean v)** Sets the bits from startIndex to endIndex, to the value passed in v. true sets the bits, false clears the bits.
24	**int size( )** Returns the number of bits in the invoking BitSet object.
25	**String toString( )** Returns the string equivalent of the invoking BitSet object.

26	**void xor(BitSet bitSet)** XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.

Example

The following program illustrates several of the methods supported by this data structure −

```
import java.util.BitSet;

public class BitSetDemo {


  public static void main(String args[]) {

     BitSet bits1 = new BitSet(16);

     BitSet bits2 = new BitSet(16);


     // set some bits

     for(int i = 0; i < 16; i++) {

        if((i % 2) == 0) bits1.set(i);

        if((i % 5) != 0) bits2.set(i);

     }


     System.out.println("Initial pattern in bits1: ");

     System.out.println(bits1);

     System.out.println("\nInitial pattern in bits2: ");

     System.out.println(bits2);


     // AND bits

     bits2.and(bits1);

     System.out.println("\nbits2 AND bits1: ");

     System.out.println(bits2);
```

```
    // OR bits
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

This will produce the following result −

Output

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

Date class

The class Date represents a specific instant in time, with millisecond precision. The Date class of java.util package implements Serializable, Cloneable and Comparable interface. It provides constructors and methods to deal with date and time with java.

Constructors

- **Date()** : Creates date object representing current date and time.

- **Date(long milliseconds)** : Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
- **Date(int year, int month, int date)**
- **Date(int year, int month, int date, int hrs, int min)**
- **Date(int year, int month, int date, int hrs, int min, int sec)**
- **Date(String s)**

Note : The last 4 constructors of the Date class are Deprecated.

```
// Java program to demonstrate constuctors of Date

import java.util.*;

  public class Main

{

    public static void main(String[] args)

    {

        Date d1 = new Date();

        System.out.println("Current date is " + d1);

        Date d2 = new Date(2323223232L);

        System.out.println("Date represented is "+ d2 );

    }

}
```

Output:

```
Current date is Tue Jul 12 18:35:37 IST 2016

Date represented is Wed Jan 28 02:50:23 IST 1970
```

Important Methods

- **boolean after(Date date) :** Tests if current date is after the given date.
- **boolean before(Date date) :** Tests if current date is before the given date.
- **int compareTo(Date date) :** Compares current date with given date. Returns 0 if the argument Date is equal to the Date; a value less than 0 if the Date is before the Date argument; and a value greater than 0 if the Date is after the Date argument.
- **long getTime()** : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
- **void setTime(long time)** : Changes the current date and time to given time.

```
// Program to demonstrate methods of Date class

import java.util.*;


public class Main
```

```
{
   public static void main(String[] args)
   {
      // Creating date
      Date d1 = new Date(2000, 11, 21);
      Date d2 = new Date();  // Current date
      Date d3 = new Date(2010, 1, 3);
      boolean a = d3.after(d1);
      System.out.println("Date d3 comes after " +  "date d2: " + a);

      boolean b = d3.before(d2);
      System.out.println("Date d3 comes before "+ "date d2: " + b);
     int c = d1.compareTo(d2);
      System.out.println(c);
     System.out.println("Miliseconds from Jan 1 "+ "1970 to date d1 is " + d1.getTime());

      System.out.println("Before setting "+d2);
      d2.setTime(204587433443L);
      System.out.println("After setting "+d2);
   }
}
```

Output:

```
Date d3 comes after date d2: true
Date d3 comes before date d2: false
1
Miliseconds from Jan 1 1970 to date d1 is 60935500800000
Before setting Tue Jul 12 13:13:16 UTC 2016
After setting Fri Jun 25 21:50:33 UTC 1976
```

Calendar Class

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

Java Calendar class declaration

Let's see the declaration of java.util.Calendar class.

```
1. public abstract class Calendar extends Object
2. implements Serializable, Cloneable, Comparable<Calendar>
```

Java Calendar Class Example

```
1. import java.util.Calendar;
2. public class CalendarExample1 {
3.   public static void main(String[] args) {
4.   Calendar calendar = Calendar.getInstance();
5.   System.out.println("The current date is : " + calendar.getTime());
6.   calendar.add(Calendar.DATE, -15);
7.   System.out.println("15 days ago: " + calendar.getTime());
8.   calendar.add(Calendar.MONTH, 4);
9.   System.out.println("4 months later: " + calendar.getTime());
10.  calendar.add(Calendar.YEAR, 2);
11.  System.out.println("2 years later: " + calendar.getTime());
12.  }
13. }
```

Output:

```
The current date is : Thu Jan 19 18:47:02 IST 2017
15 days ago: Wed Jan 04 18:47:02 IST 2017
4 months later: Thu May 04 18:47:02 IST 2017
2 years later: Sat May 04 18:47:02 IST 2019
```

Java Calendar Class Example: get()

```
1. import java.util.*;
2. public class CalendarExample2{
3.  public static void main(String[] args) {
4.   Calendar calendar = Calendar.getInstance();
5.   System.out.println("At present Calendar's Year: " + calendar.get(Calendar.YEAR));
6.   System.out.println("At present Calendar's Day: " + calendar.get(Calendar.DATE));
7.   }
```

```
}
```

Output:

```
At present Calendar's Year: 2017
At present Calendar's Day: 20
```

Java Calendar Class Example: getInstance()

```
import java.util.*;
public class CalendarExample3{
  public static void main(String[] args) {
  Calendar calendar = Calendar.getInstance();
  System.out.print("At present Date And Time Is: " + calendar.getTime());
  }
}
```

Output:

```
At present Date And Time Is: Fri Jan 20 14:26:19 IST 2017
```

Java Calendar Class Example: getMaximum()

```
import java.util.*;
public class CalendarExample4 {
  public static void main(String[] args) {
  Calendar calendar = Calendar.getInstance();
  int maximum = calendar.getMaximum(Calendar.DAY_OF_WEEK);
  System.out.println("Maximum number of days in week: " + maximum);
  maximum = calendar.getMaximum(Calendar.WEEK_OF_YEAR);
  System.out.println("Maximum number of weeks in year: " + maximum);
  }
}
```

Output:

```
Maximum number of days in week: 7
Maximum number of weeks in year: 53
```

Java Calendar Class Example: getMinimum()

```
import java.util.*;
public class CalendarExample5 {
```

```
3.    public static void main(String[] args) {
4.    Calendar cal = Calendar.getInstance();
5.    int maximum = cal.getMinimum(Calendar.DAY_OF_WEEK);
6.    System.out.println("Minimum number of days in week: " + maximum);
7.    maximum = cal.getMinimum(Calendar.WEEK_OF_YEAR);
8.    System.out.println("Minimum number of weeks in year: " + maximum);
9.    }
10. }
```

Output:

```
Minimum number of days in week: 1
Minimum number of weeks in year: 1
```

Java Random class

Java Random class is used to generate a stream of pseudorandom numbers. The algorithms implemented by Random class use a protected utility method than can supply up to 32 pseudo randomly generated bits on each invocation.

Example 1

```
1.  import java.util.Random;
2.  public class JavaRandomExample1 {
3.     public static void main(String[] args) {
4.         //create random object
5.         Random random= new Random();
6.         //returns unlimited stream of pseudorandom long values
7.             System.out.println("Longs value : "+random.longs());
8.        // Returns the next pseudorandom boolean value
9.         boolean val = random.nextBoolean();
10.        System.out.println("Random boolean value : "+val);
11.        byte[] bytes = new byte[10];
12.        //generates random bytes and put them in an array
13.        random.nextBytes(bytes);
14.        System.out.print("Random bytes = ( ");
15.        for(int i = 0; i< bytes.length; i++)
16.        {
17.            System.out.printf("%d ", bytes[i]);
18.        }
19.        System.out.print(")");
20.    }
21. }
```

Output:

Longs value : java.util.stream.LongPipeline$Head@14ae5a5
Random boolean value : true
Random bytes = (57 77 8 67 -122 -71 -79 -62 53 19)

Example 2

```
1. import java.util.Random;
2. public class JavaRandomExample2 {
3.   public static void main(String[] args) {
4.     Random random = new Random();
5.     //return the next pseudorandom integer value
6.     System.out.println("Random Integer value : "+random.nextInt());
7.     // setting seed
8.     long seed =20;
9.     random.setSeed(seed);
10.    //value after setting seed
11.    System.out.println("Seed value : "+random.nextInt());
12.    //return the next pseudorandom long value
13.    Long val = random.nextLong();
14.    System.out.println("Random Long value : "+val);
15.  }
16. }
```

Output:

Random Integer value : 1294094433
Seed value : -1150867590
Random Long value : -7322354119883315205

Formatter class

Formatter class outputs the formatted output. It can format numbers, strings, and time and date. It operates in a manner similar to the C/C++ printf() function.

Formatting Basics

The most commonly used format function is:

Formatter format(String fmtString, Object ... args)

The fmtSring consists characters that are copied to the output buffer and the format specifiers.

A format specifier begins with a percent sign followed by the format conversion specifier. For example, the format specifier for floating-point data is

%f

The format specifiers and the arguments are matched in order from left to right. For example, consider this fragment:

```
import java.util.Formatter;
public class Main {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    fmt.format("%s gap filler %d %f", "Astring", 10, 12.3);
    System.out.println(fmt.out());
  }
}
```

The output:

```
Astring gap filler 10 12.300000
```

In this example, the format specifiers, %s, %d, and %f, are replaced with the arguments that follow the format string. %s is replaced by "Astring", %d is replaced by 10, and %f is replaced by 12.3. %s specifies a string, and %d specifies an integer value. %f specifies a floating-point value. All other characters are simply used as-is.

Format specifier

The format() method accepts a wide variety of format specifiers. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and lowercase specifiers perform the same conversion. The following table shows the format specifiers:

Format Specifier	Conversion Applied
%a %A	Floating-point hexadecimal
%b %B	Boolean
%c	Character
%d	Decimal integer
%h %H	Hash code of the argument
%e %E	Scientific notation
%f	Decimal floating-point
%g %G	Uses %e or %f, whichever is shorter
%o	Octal integer
%n	Inserts a newline character
%s %S	String
%t %T	Time and date
%x %X	Integer hexadecimal
%%	Inserts a % sign

If the argument doesn't match the type-checks, an IllegalFormatException is thrown.

Formatting Strings and Characters

To format an individual character, use %c. This causes the matching character argument to be output, unmodified.

To format a string, use %s.

format('%.15s', String str) (Display at most 15 characters in a string)

The following code use %s specifier and number to control the output width. It will only display at most 15 characters in a string.

```
import java.util.Formatter;
//from  j  a v a2s.  c o  m
public class MainClass {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt = new Formatter();
    fmt.format("%.15s", "Formatting with Java is now easy.");
    System.out.println(fmt);
  }
}
```

The output:

```
  Formatting with
```

Pass in two strings

```
public class Main {
  public static void main(String args[]) {
//from j  a  va 2 s  .c o  m
    System.out.println(
      String.format("Hi %s at %s", "Formatter example", "your screen"));
  }
}
```

The code above generates the following result.

```
Hi Formatter example at your screen
```

Java Scanner Class

Java Scanner class comes under the java.util package. Java has various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using a regular expression.

Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

Java Scanner Class Declaration

```
1. public final class Scanner
2.         extends Object
3.         implements Iterator<String>
```

Java Scanner Class Constructors

SN	Constructor	Description
1)	Scanner(File source)	It constructs a new Scanner that produces values scanned from the specified file.
2)	Scanner(File source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.
3)	Scanner(InputStream source)	It constructs a new Scanner that produces values scanned from the specified input stream.
4)	Scanner(InputStream source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified input stream.

5)	Scanner(Readable source)	It constructs a new Scanner that produces values scanned from the specified source.
6)	Scanner(String source)	It constructs a new Scanner that produces values scanned from the specified string.
7)	Scanner(ReadableByteChannel source)	It constructs a new Scanner that produces values scanned from the specified channel.
8)	Scanner(ReadableByteChannel source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified channel.
9)	Scanner(Path source)	It constructs a new Scanner that produces values scanned from the specified file.
10)	Scanner(Path source, String charsetName)	It constructs a new Scanner that produces values scanned from the specified file.

Java Scanner Class Methods

The following are the list of Scanner methods:

SN	Modifier & Type	Method	Description
1)	Void	close()	It is used to close this scanner.
2)	pattern	delimiter()	It is used to get the Pattern which the Scanner class is currently using to match

			delimiters.
3)	Stream<MatchResult>	findAll()	It is used to find a stream of match results that match the provided pattern string.
4)	String	findInLine()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string	findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean	hasNext()	It returns true if this scanner has another token in its input.
7)	boolean	hasNextBigDecimal()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
8)	boolean	hasNextBigInteger()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextBigDecimal() method or not.
9)	boolean	hasNextBoolean()	It is used to check if the next token in this scanner's

			input can be interpreted as a Boolean using the nextBoolean() method or not.
10)	boolean	hasNextByte()	It is used to check if the next token in this scanner's input can be interpreted as a Byte using the nextBigDecimal() method or not.
11)	boolean	hasNextDouble()	It is used to check if the next token in this scanner's input can be interpreted as a BigDecimal using the nextByte() method or not.
12)	boolean	hasNextFloat()	It is used to check if the next token in this scanner's input can be interpreted as a Float using the nextFloat() method or not.
13)	boolean	hasNextInt()	It is used to check if the next token in this scanner's input can be interpreted as an int using the nextInt() method or not.
14)	boolean	hasNextLine()	It is used to check if there is another line in the input of this scanner or not.
15)	boolean	hasNextLong()	It is used to check if the next token in this scanner's input can be interpreted as a Long using the nextLong()

			method or not.
16)	boolean	hasNextShort()	It is used to check if the next token in this scanner's input can be interpreted as a Short using the nextShort() method or not.
17)	IOException	ioException()	It is used to get the IOException last thrown by this Scanner's readable.
18)	Locale	locale()	It is used to get a Locale of the Scanner class.
19)	MatchResult	match()	It is used to get the match result of the last scanning operation performed by this scanner.
20)	String	next()	It is used to get the next complete token from the scanner which is in use.
21)	BigDecimal	nextBigDecimal()	It scans the next token of the input as a BigDecimal.
22)	BigInteger	nextBigInteger()	It scans the next token of the input as a BigInteger.
23)	boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
24)	Byte	nextByte()	It scans the next token of the input as a byte.

25)	double	nextDouble()	It scans the next token of the input as a double.
26)	Float	nextFloat()	It scans the next token of the input as a float.
27)	Int	nextInt()	It scans the next token of the input as an Int.
28)	String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
29)	Long	nextLong()	It scans the next token of the input as a long.
30)	short	nextShort()	It scans the next token of the input as a short.
31)	Int	radix()	It is used to get the default radix of the Scanner use.
32)	Void	remove()	It is used when remove operation is not supported by this implementation of Iterator.
33)	Scanner	reset()	It is used to reset the Scanner which is in use.
34)	Scanner	skip()	It skips input that matches the specified pattern, ignoring delimiters
35)	Stream<String>	tokens()	It is used to get a stream of delimiter-separated tokens from the Scanner object

			which is in use.
36)	String	toString()	It is used to get the string representation of Scanner using.
37)	Scanner	useDelimiter()	It is used to set the delimiting pattern of the Scanner which is in use to the specified pattern.
38)	Scanner	useLocale()	It is used to sets this scanner's locale object to the specified locale.
39)	Scanner	useRadix()	It is used to set the default radix of the Scanner which is in use to the specified radix.

Example 1

```
1. import java.util.*;
2. public class ScannerClassExample1 {
3.     public static void main(String args[]){
4.         String s = "Hello, This is JavaTpoint.";
5.         //Create scanner Object and pass string in it
6.         Scanner scan = new Scanner(s);
7.         //Check if the scanner has a token
8.         System.out.println("Boolean Result: " + scan.hasNext());
9.         //Print the string
10.        System.out.println("String: " +scan.nextLine());
11.        scan.close();
12.        System.out.println("--------Enter Your Details-------- ");
13.        Scanner in = new Scanner(System.in);
14.        System.out.print("Enter your name: ");
15.        String name = in.next();
16.        System.out.println("Name: " + name);
```

```
17.         System.out.print("Enter your age: ");
18.         int i = in.nextInt();
19.         System.out.println("Age: " + i);
20.         System.out.print("Enter your salary: ");
21.         double d = in.nextDouble();
22.         System.out.println("Salary: " + d);
23.         in.close();
24.         }
25. }
```

Output:

```
Boolean Result: true
String: Hello, This is JavaTpoint.
-------Enter Your Details---------
Enter your name: Abhishek
Name: Abhishek
Enter your age: 23
Age: 23
Enter your salary: 25000
Salary: 25000.0
```