Java Collection Framework

Java collection framework represents a hierarchy of set of interfaces and classes that are used to manipulate group of objects.

Collections framework was added to Java 1.2 version. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects.

Java collections framework is contained in java.util package. It provides many important classes and interfaces to collect and organize group of objects.

What is collection

Collection in java can be referred to an object that collects multiple elements into a single unit. It is used to store, fetch and manipulate data. For example, list is used to collect elements and referred by a list object.

Components of Collection Framework

The collections framework consists of:

- Collection interfaces such as sets, lists, and maps. These are used to collect different types of objects.
- Collection classes such as ArrayList, HashSet etc that are implementations of collection interfaces.
- Concurrent implementation classes that are designed for highly concurrent use.
- Algorithms that provides static methods to perform useful functions on collections, such as sorting a list.

Advantage of Collection Framework

- It reduces programming effort by providing built-in set of data structures and algorithms.
- Increases performance by providing high-performance implementations of data structures and algorithms.
- Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.
- Increase Productivity
- Reduce operational time
- versatility to work with current collection as well

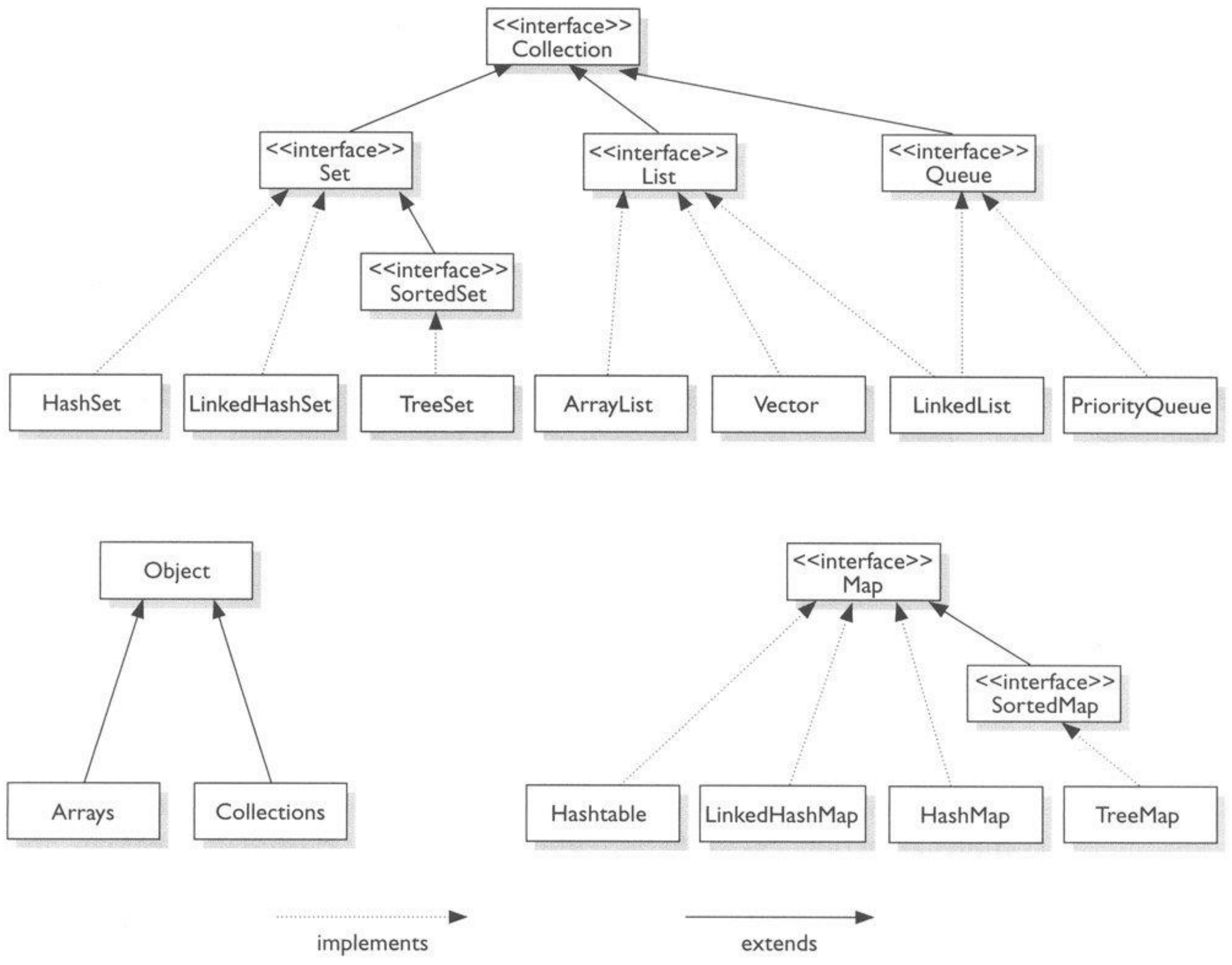
Important Interfaces of Collection API

Interface	**Description**
Collection	Enables you to work with groups of object; it is at the top of Collection hierarchy
Deque	Extends Queue to handle double ended queue.
List	Extends Collection to handle sequences list of object.

Queue	Extends Collection to handle special kind of list in which element are removed only from the head.
Set	Extends Collection to handle sets, which must contain unique element.
SortedSet	Extends Set to handle sorted set.

Java Collection Hierarchy

Collection Framework hierarchy is represented by using a diagram. You can get idea how interfaces and classes are linked with each other and in what hierarchy they are situated. Top of the framework, Collection framework is available and rest of interfaces are sub interface of it.



Collection Heirarchy

All these Interfaces give several methods which are defined by collections classes which implement these interfaces.

Commonly Used Methods of Collection Framework

Method	Description

public boolean add(E e)	It inserts an element in this collection.
public boolean addAll(Collection<? extends E> c)	It inserts the specified collection elements in the invoking collection.
public boolean remove(Object element)	It deletes an element from the collection.
public boolean removeAll(Collection<?> c)	It deletes all the elements of the specified collection from the invoking collection.
default boolean removeIf(Predicate<? super E> filter)	It deletes all the elements of the collection that satisfy the specified predicate.
public boolean retainAll(Collection<?> c)	It deletes all the elements of invoking collection except the specified collection.
public int size()	It returns the total number of elements in the collection.
public void clear()	It removes the total number of elements from the collection.
public boolean contains(Object element)	It searches for an element.
public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.

Why Collections were made Generic?

Generics added **type safety** to Collection framework. Earlier collections stored **Object class** references which meant any collection could store any type of object. Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch. Hence Generics was introduced through which you can explicitly state the type of object being stored.

Collections and Autoboxing

We have studied that Autoboxing converts primitive types into Wrapper class Objects. As collections doesn't store primitive data types(stores only refrences), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

Most Commonly thrown Exceptions in Collections Framework

There may be chance of getting exceptions while working with collections. We have listed some most common exceptions that may occur during program execution.

Exception Name	Description
UnSupportedOperationException	occurs if a Collection cannot be modified
ClassCastException	occurs when one object is incompatible with another
NullPointerException	occurs when you try to store null object in Collection
IllegalArgumentException	thrown if an invalid argument is used
IllegalStateException	thrown if you try to add an element to an already full Collection

Interfaces of Java Collection Framework

The Collections framework has a lot of Interfaces, setting the fundamental nature of various collection classes. Lets study the most important Interfaces in the Collections framework.

The Collection Interface

It is at the top of collection heirarchy and must be implemented by any class that defines a collection. Its general declaration is,

```
interface Collection <E>
```

Copy

Collection Interface Methods

1. Following are some of the commonly used methods in this interface.

Methods	**Description**
boolean add(E obj)	Used to add objects to a collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
boolean addAll(Collection C)	Add all elements of collection C to the invoking collection. Returns true if the element were added. Otherwise, returns false.
boolean remove(Object obj)	To remove an object from collection. Returns true if the element was removed. Otherwise, returns false.
boolean removeAll(Collection C)	Removes all element of collection C from the invoking collection. Returns true if the collection's elements were removed. Otherwise, returns false.
boolean contains(Object obj)	To determine whether an object is present in collection or not. Returns true if obj is an element of the invoking collection. Otherwise, returns false.
boolean isEmpty()	Returns true if collection is empty, else returns false.
int size()	Returns number of elements present in collection.
void clear()	Removes total number of elements from the collection.

Object[] toArray()	Returns an array which consists of the invoking collection elements.
boolean retainAll(Collection c)	Deletes all the elements of invoking collection except the specified collection.
Iterator iterator( )	Returns an iterator for the invoking collection.
boolean equals(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
Object[] toArray(Object array[])	Returns an array containing only those collection elements whose type matches of the specified array.

The List Interface

It extends the **Collection** Interface, and defines storage as sequence of elements. Following is its general declaration,

```
interface List <E>
```

Copy

1. Allows random access and insertion, based on position.
2. It allows Duplicate elements.

   List Interface Methods

3. Apart from methods of Collection Interface, it adds following methods of its own.

Methods	**Description**
Object get(int index)	Returns object stored at the specified index
Object set(int index, E obj)	Stores object at the specified index in the calling collection
int indexOf(Object obj)	Returns index of first occurrence of obj in the collection
int lastIndexOf(Object obj)	Returns index of last occurrence of obj in the collection
List subList(int start, int end)	Returns a list containing elements between start and end index in the collection

The Set Interface

This interface defines a Set. It extends **Collection** interface and doesn't allow insertion of duplicate elements. It's general declaration is,

```
interface Set <E>
```

Copy

1. It doesn't define any method of its own. It has two sub interfaces, **SortedSet** and **NavigableSet**.
2. **SortedSet** interface extends **Set** interface and arranges added elements in an ascending order.
3. **NavigabeSet** interface extends **SortedSet** interface, and allows retrieval of elements based on the closest match to a given value or values.

The Queue Interface

It extends **collection** interface and defines behaviour of queue, that is first-in, first-out. It's general declaration is,

```
interface Queue <E>
```

Copy

Queue Interface Methods

There are couple of new and interesting methods added by this interface. Some of them are mentioned in below table.

Methods	Description
Object poll()	removes element at the head of the queue and returns **null** if queue is empty
Object remove()	removes element at the head of the queue and throws **NoSuchElementException** if queue is empty
Object peek()	returns the element at the head of the queue without removing it. Returns **null** if queue is empty
Object element()	same as peek(), but throws **NoSuchElementException** if queue is empty
boolean offer(E obj)	Adds object to queue.

The Dequeue Interface

It extends **Queue** interface and implements behaviour of a double-ended queue. Its general declaration is,

```
interface Dequeue <E>
```

Copy

1. Since it implements Queue interface, it has the same methods as mentioned there.
2. Double ended queues can function as simple queues as well as like standard Stacks.

The Collection classes in Java

Java collection framework consists of various classes that are used to store objects. These classes on the top implements the Collection interface. Some of the classes provide full implementations that can be used as it is. Others are abstract classes, which provides skeletal implementations that can be used as a starting point for creating concrete collections.

Java Collection Framework Classes

This table contains abstract and non-abstract classes that implements collection interface.

The standard collection classes are:

Class	Description
AbstractCollection	Implements most of the Collection interface.
AbstractList	Extends AbstractCollection and implements most of the List interface.
AbstractQueue	Extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending AbstractSequentialList
ArrayList	Implements a dynamic array by extending AbstractList
ArrayDeque	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface(Added by Java SE 6).
AbstractSet	Extends AbstractCollection and implements most of the Set interface.

EnumSet	Extends AbstractSet for use with enum elements.
HashSet	Extends AbstractSet for use with a hash table.
LinkedHashSet	Extends HashSet to allow insertion-order iterations.
PriorityQueue	Extends AbstractQueue to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends AbstractSet.

Note:

1. To use any Collection class in your program, you need to import java.util package.
2. Whenever you print any Collection class, it gets printed inside the square brackets [] with its elements.

ArrayList class

This class provides implementation of an array based data structure that is used to store elements in linear order. This class implements List interface and an abstract AbstractList class. It creates a dynamic array that grows based on the elements strength.

Simple array has fixed size i.e it can store fixed number of elements but sometimes you may not know beforehand about the number of elements that you are going to store in your array. In such situations, We can use an ArrayList, which is an array whose size can increase or decrease dynamically.

1. ArrayList class extends **AbstractList** class and implements the **List** interface.
2. ArrayList supports dynamic array that can grow as needed.
3. It can contain Duplicate elements and it also maintains the insertion order.
4. Manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.
5. ArrayLists are not synchronized.
6. ArrayList allows random access because it works on the index basis.

ArrayList Constructors

ArrayList class has three constructors that can be used to create Arraylist either from empty or elements of other collection.

```
ArrayList()  // It creates an empty ArrayList

ArrayList( Collection C ) // It creates an ArrayList that is
initialized with elements of the Collection C

ArrayList( int capacity ) // It creates an ArrayList that has the
specified initial capacity
```

Copy

Example of ArrayList

Lets create an ArrayList to store string elements. See, we used add method of list interface to add elements.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

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

    al.add("ab");

    al.add("bc");

    al.add("cd");

    System.out.println(al);

  }

}
```

Copy

```
[ab,bc,cd]
```

LinkedList class

Java LinkedList class provides implementation of linked-list data structure. It used doubly linked list to store the elements.

1. LinkedList class extends AbstractSequentialList and implements List,Deque and Queue inteface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It is dynamic in nature i.e it allocates memory when required. Therefore insertion and deletion operations can be easily implemented.
4. It can contain duplicate elements and it is not synchronized.

5. Reverse Traversing is difficult in linked list.
6. In LinkedList, manipulation is fast because no shifting needs to be occurred.

LinkedList Constructors

LinkedList class has two constructors.

```
LinkedList() // It creates an empty LinkedList

LinkedList( Collection c) // It creates a LinkedList that is
initialized with elements of the Collection c
```

Copy

LinkedList class Example

Lets take an example to create a linked-list and add elements using add and other methods as well. See the below example.

```
import java.util.* ;

class Demo

{

  public static void main(String[] args)

  {

    LinkedList< String> ll = new LinkedList< String>();

    ll.add("a");

    ll.add("b");

    ll.add("c");

    ll.addLast("z");

    ll.addFirst("A");

    System.out.println(ll);

  }

}
```

Copy

```
[A, a, b,c, z]
```

Difference between ArrayList and Linked List

ArrayList and **LinkedList** are the Collection classes, and both of them implements the List interface. The ArrayList class creates the list which is internally stored in a dynamic array that grows or shrinks in size as the elements are added or deleted from it. LinkedList also creates the list which is internally stored in a DoublyLinked List. Both the classes are used to store the elements in the list, but the major difference between both the classes is that ArrayList allows random access to the elements in the list as it operates on an **index-based** data structure. On the other hand, the LinkedList does not allow random access as it does not have indexes to access elements directly, it has to traverse the list to retrieve or access an element from the list.

Some more differences:

- ArrayList extends AbstarctList class whereas LinkedList extends AbstractSequentialList.
- AbstractList implements List interface, thus it can behave as a list only whereas LinkedList implements List, Deque and Queue interface, thus it can behave as a Queue and List both.
- In a list, access to elements is faster in ArrayList as random access is also possible. Access to LinkedList elements is slower as it follows sequential access only.
- In a list, manipulation of elements is slower in ArrayList whereas it is faster in LinkedList.

HashSet class

1. HashSet extends **AbstractSet** class and implements the **Set** interface.
2. HashSet has three constructors.

```
HashSet()  //This creates an empty HashSet



HashSet( Collection C )  //This creates a HashSet that is
initialized with the elements of the Collection C



HashSet( int capacity )  //This creates a HashSet that has the
specified initial capacity
```

   Copy

3. It creates a collection that uses hash table for storage. A hash table stores information by using a mechanism called **hashing**.
4. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored.
5. HashSet does not maintain any order of elements.
6. HashSet contains only unique elements.

Example of HashSet class

In this example, we are creating a HashSet that store string values. Since HashSet does not store duplicate elements, we tried to add a duplicate elements but the output contains only unique elements.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

    HashSet<String> hs = new HashSet<String>();

    hs.add("B");

    hs.add("A");

    hs.add("D");

    hs.add("E");

    hs.add("C");

    hs.add("A");

    System.out.println(hs);

  }

}
```

Copy

```
[A, B, C, D, E]
```

LinkedHashSet Class

1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet maintains a linked list of entries in the set.
3. LinkedHashSet stores elements in the order in which elements are inserted i.e it maintains the insertion order.

Example of LinkedHashSet class

```
import java.util.*;

class Demo

{
  public static void main(String args[])

  {

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

    hs.add("B");

    hs.add("A");

    hs.add("D");

    hs.add("E");

    hs.add("C");

    hs.add("F");

    System.out.println(hs);

  }

}
```

Copy

```
[B, A, D, E, C, F]
```

TreeSet Class

1. It extends **AbstractSet** class and implements the **NavigableSet** interface.
2. It stores the elements in ascending order.
3. It uses a Tree structure to store elements.
4. It contains unique elements only like HashSet.
5. It's access and retrieval times are quite fast.
6. It has four Constructors.

```
TreeSet()  //It creates 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 creates a new tree set that contains
the elements of the Collection C



TreeSet( Comparator comp )  //It creates an empty tree set that
will be sorted according to given Comparator
```

```
TreeSet( SortedSet ss )  //It creates a TreeSet that contains the
elements of given SortedSet
```

Example of TreeSet class

Lets take an example to create a treeset that contains duplicate elements. But you can notice that it prints unique elements that means it does not allow duplicate elements.

```
import java.util.*;

class Demo{

 public static void main(String args[]){

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

   al.add("Ravi");

   al.add("Vijay");

   al.add("Ravi");

   al.add("Ajay");


  Iterator itr=al.iterator();

  while(itr.hasNext()){

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

  }

 }

}
```
Copy

```
Ajay

Ravi

Vijay
```

PriorityQueue Class

1. It extends the **AbstractQueue** class.
2. The PriorityQueue class provides the facility of using queue.
3. It does not orders the elements in FIFO manner.
4. PriorityQueue has six constructors. In all cases, the capacity grows automatically as elements are added.

```
5. PriorityQueue( )  //This constructor creates an empty queue. By
  default, its starting capacity is 11
```

```
PriorityQueue(int capacity) //This constructor creates a queue that has the specified initial capacity

PriorityQueue(int capacity, Comparator comp) //This constructor creates a queue with the specified capacity
and comparator

//The last three constructors create queues that are initialized with elements of Collection passed in c
PriorityQueue(Collection c)

PriorityQueue(PriorityQueue c)

PriorityQueue(SortedSet c)
```

Copy

Note: If no comparator is specified when a PriorityQueue is constructed, then the default comparator for the type of data stored in the queue is used. The default comparator will order the queue in ascending order. Thus, the head of the queue will be the smallest value. However, by providing a custom comparator, you can specify a different ordering scheme.

Example of PriorityQueue class

Lets take an example to create a priority queue that store and remove elements.

```
import java.util.*;


class Demo

{

  public static void main(String args[])

  {

    PriorityQueue<String> queue=new PriorityQueue<String>();

    queue.add("WE");

    queue.add("LOVE");

    queue.add("STUDY");

    queue.add("TONIGHT");
    System.out.println("At head of the queue:"+queue.element());

    System.out.println("At head of the queue:"+queue.peek());

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

    Iterator itr=queue.iterator();

    while(itr.hasNext()){

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

    }

    queue.remove();

    queue.poll();

    System.out.println("After removing two elements:");

    Iterator itr2=queue.iterator();

    while(itr2.hasNext()){

      System.out.println(itr2.next());

    }

  }

}
```

Copy

```
At head of the queue:LOVE

At head of the queue:LOVE

Iterating the queue elements:

LOVE

TONIGHT

STUDY

WE

After removing two elements:

TONIGHT

WE
```

Accessing a Java Collection using Iterators

To access elements of a collection, either we can use index if collection is list based or we need to traverse the element. There are three possible ways to traverse through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

Accessing elements using Iterator

Iterator is an interface that is used to iterate the collection elements. It is part of java collection framework. It provides some methods that are used to check and access elements of a collection.

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide `iterator()` method to return an iterator.

Iterator Interface Methods

Method	Description
boolean `hasNext()`	Returns **true** if there are more elements in the collection. Otherwise, returns false.
E `next()`	Returns the **next element present** in the collection. Throws NoSuchElementException if there is not a next element.
void `remove()`	Removes the current element. Throws IllegalStateException if an attempt is made to call remove() method that is not preceded by a call to `next()` method.

Iterator Example

In this example, we are using iterator() method of collection interface that returns an instance of Iterator interface. After that we are using hasNext() method that returns true of collection contains an elements and within the loop, obtain each element by calling `next()` method.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)
  {
    ArrayList< String> ar = new ArrayList< String>();
    ar.add("ab");
    ar.add("bc");
    ar.add("cd");
    ar.add("de");
    Iterator it = ar.iterator();     //Declaring Iterator
    while(it.hasNext())
    {
      System.out.print(it.next()+" ");
    }
  }
}
```

Copy

```
ab bc cd de
```

Accessing elements using ListIterator

ListIterator Interface is used to traverse a list in both **forward** and **backward** direction. It is available to only those collections that implements the **List** Interface.

Methods of ListIterator:

Method	Description
void add(E obj)	Inserts obj into the list in front of the element that will be returned by the next call to next() method.
boolean hasNext()	Returns true if there is a next element. Otherwise, returns false.
boolean hasPrevious()	Returns true if there is a previous element. Otherwise, returns false.

E next()	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
int nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous()	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
int previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.
void remove()	Removes the current element from the list. An IllegalStateException is thrown if remove() method is called before next() or previous() method is invoked.
void set(E obj)	Assigns obj to the current element. This is the element last returned by a call to either next() or previous() method.

ListIterator Example

Lets create an example to traverse the elements of ArrayList. ListIterator works only with list collection.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

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

    ar.add("ab");

    ar.add("bc");

    ar.add("cd");

    ar.add("de");
    ListIterator litr = ar.listIterator();

    while(litr.hasNext())    //In forward direction

    {

      System.out.print(litr.next()+"  ");

    }

    while(litr.hasPrevious())    //In backward direction

    {

      System.out.print(litr.previous()+"  ");

    }

  }

}
```

Copy

```
ab bc cd de de cd bc ab
```

for-each loop

`for-each` version of `for` loop can also be used for traversing the elements of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any **reverse** access. `for-each` loop can cycle through any collection of object that implements `Iterable` interface.

Exmaple:

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    LinkedList< String> ls = new LinkedList< String>();

    ls.add("a");

    ls.add("b");

    ls.add("c");

    ls.add("d");

    for(String str : ls)

    {
      System.out.print(str+" ");
    }
  }
}
```

Copy

```
a b c d
```

Traversing using for loop

we can use for loop to traverse the collection elements but only index-based collection can be accessed. For example, list is index-based collection that allows to access its elements using the index value.

```
import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    LinkedList<String> ls = new LinkedList<String>();
    ls.add("a");
    ls.add("b");
    ls.add("c");
    ls.add("d");
    for(int i = 0; i<ls.size(); i++)
    {
      System.out.print(ls.get(i)+" ");
    }
  }
}
```

Copy

```
Output-
a b c d
```

Java Collection Framework ArrayList

This class provides implementation of an array based data structure that is used to store elements in linear order. This class implements List interface and an abstract AbstractList class. It creates a dynamic array that grows based on the elements strength.

Java ArrayList class Declaration

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

Copy

1. ArrayList supports dynamic array that can grow as needed.
2. It can contain Duplicate elements and it also maintains the insertion order.
3. Manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.
4. ArrayLists are not synchronized.
5. ArrayList allows random access because it works on the index basis.

ArrayList Constructors

ArrayList class has three constructors that can be used to create Arraylist either from empty or elements of other collection.

```
ArrayList()  // It creates an empty ArrayList

ArrayList( Collection C ) // It creates an ArrayList that is
initialized with elements of the Collection C

ArrayList( int capacity ) // It creates an ArrayList that has the
specified initial capacity
```

Copy

Example: Creating an ArrayList

Lets create an ArrayList to store string elements. Here list is empty because we did not add elements to it.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)
  {

    // Creating an ArrayList

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


    // Displaying Arraylist

    System.out.println(fruits);

  }

}
```

Copy

```
[]
```

ArrayList Methods

The below table contains methods of Arraylist. We can use them to manipulate its elements.

Method	Description
void add(int index, E element)	It inserts the specified element at the specified position in a list.
boolean add(E e)	It appends the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It appends all of the elements in the specified collection to the end of this list.
boolean addAll(int index, Collection<? extends E> c)	It appends all the elements in the specified collection, starting at the specified position of the list.
void clear()	It removes all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It enhances the capacity of an ArrayList instance.

E get(int index)	It fetches the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It returns 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 returns an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It returns an array containing all of the elements in this list in the correct order.
Object clone()	It returns a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It removes the element present at the specified position in the list.
boolean remove(Object o)	It removes the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It removes all the elements from the list.

boolean removeIf(Predicate<? super E> filter)	It removes all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It removes all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It replaces all the elements from the list with the specified element.
void trimToSize()	It trims the capacity of this ArrayList instance to be the list's current size.

Add Elements to ArrayList

To add elements into ArrayList, we are using add() method. It adds elements into the list in the insertion order.

```
import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    // Creating an ArrayList
    ArrayList< String> fruits = new ArrayList< String>();
    // Adding elements to ArrayList
    fruits.add("Mango");
    fruits.add("Apple");
    fruits.add("Berry");
    // Displaying ArrayList
    System.out.println(fruits);
  }
}
```

Copy

```
[Mango, Apple, Berry]
```

Removing Elements

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating an ArrayList

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

    // Adding elements to ArrayList

    fruits.add("Mango");

    fruits.add("Apple");

    fruits.add("Berry");

    fruits.add("Orange");

    // Displaying ArrayList

    System.out.println(fruits);

    // Removing Elements

    fruits.remove("Apple");

    System.out.println("After Deleting Elements: \n"+fruits);

    // Removing Second Element

    fruits.remove(1);

    System.out.println("After Deleting Elements: \n"+fruits);


  }

}
```

Copy

```
[Mango, Apple, Berry, Orange]
After Deleting Elements:
[Mango, Berry, Orange]
After Deleting Elements:
[Mango, Orange]
```

Traversing Elements of ArrayList

Since ArrayList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the below example.

```
import java.util.*;
class Demo
{
  public static void main(String[] args)
  {
    // Creating an ArrayList
    ArrayList< String> fruits = new ArrayList< String>();
    // Adding elements to ArrayList
    fruits.add("Mango");
    fruits.add("Apple");
    fruits.add("Berry");
    fruits.add("Orange");
    // Traversing ArrayList
    for(String element : fruits) {
      System.out.println(element);
    }
  }
}
```

Copy

```
Mango
Apple
Berry
Orange
```

Get size of ArrayList

Sometimes we want to know number of elements an ArrayList holds. In that case we use size() then returns size of ArrayList which is equal to number of elements present in the list.

```java
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating an ArrayList

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

    // Adding elements to ArrayList

    fruits.add("Mango");

    fruits.add("Apple");

    fruits.add("Berry");

    fruits.add("Orange");

    // Traversing ArrayList

    for(String element : fruits) {

      System.out.println(element);

    }

    System.out.println("Total Elements: "+fruits.size());

  }

}
```

Copy

```
Mango
Apple
Berry
```

```
Orange

Total Elements: 4
```

Sorting ArrayList Elements

To sort elements of an ArrayList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort method to sort the elements.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating an ArrayList

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

    // Adding elements to ArrayList

    fruits.add("Mango");

    fruits.add("Apple");

    fruits.add("Berry");

    fruits.add("Orange");

    // Sorting elements

    Collections.sort(fruits);

    // Traversing ArrayList

    for(String element : fruits) {

      System.out.println(element);

    }

  }

}
```

Copy

```
OUTPUT-

Apple

Berry

Mango

Orange
```

Java Collection Framework Linked list

Java LinkedList class provides implementation of linked-list data structure. It used doubly linked list to store the elements. It implements List, Deque and Queue interface and extends AbstractSequentialList class. below is the declaration of LinkedList class.

LinkedList class Declaration

```
public class LinkedList<E> extends AbstractSequentialList<E> implements
List<E>, Deque<E>, Cloneable, Serializable
```

Copy

1. LinkedList class extends AbstractSequentialList and implements List,Deque and Queue inteface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It allows null entry.
4. It is dynamic in nature i.e it allocates memory when required. Therefore insertion and deletion operations can be easily implemented.
5. It can contain duplicate elements and it is not synchronized.
6. Reverse Traversing is difficult in linked list.
7. In LinkedList, manipulation is fast because no shifting needs to be occurred.

LinkedList Constructors

LinkedList class has two constructors. First is used to create empty LinkedList and second for creating from existing collection.

```
LinkedList() // It creates an empty LinkedList

LinkedList( Collection c) // It creates a LinkedList that is
initialized with elements of the Collection c
```

Copy

LinkedList class Example

Lets take an example to create a linked-list, no element is inserted so it creates an empty LinkedList. See the below example.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)
  {

    // Creating LinkedList

    LinkedList< String> linkedList = new LinkedList< String>();

    // Displaying LinkedLIst

    System.out.println(linkedList);

  }

}
```

Copy

```
[]
```

LinkedList Methods

The below table contains methods of LinkedList. We can use them to manipulate its elements.

Method	Description
boolean add(E e)	It appends the specified element to the end of a list.
void add(int index, E element)	It inserts the specified element at the specified position index in a list.
boolean addAll(Collection<? extends E> c)	It appends all of the elements in the specified collection to the end of this list.
boolean addAll(Collection<? extends E> c)	It appends 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.
boolean addAll(int index, Collection<? extends E> c)	It appends all the elements in the specified collection, starting at the specified position of the list.

void addFirst(E e)	It inserts the given element at the beginning of a list.
void addLast(E e)	It appends the given element to the end of a list.
void clear()	It removes all the elements from a list.
Object clone()	It returns a shallow copy of an ArrayList.
boolean contains(Object o)	It returns true if a list contains a specified element.
Iterator<E> descendingIterator()	It returns an iterator over the elements in a deque in reverse sequential order.
E element()	It retrieves the first element of a list.
E get(int index)	It returns the element at the specified position in a list.
E getFirst()	It returns the first element in a list.
E getLast()	It returns the last element in a list.
int indexOf(Object o)	It returns 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.
ListIterator<E> listIterator(int index)	It returns a list-iterator of the elements in proper sequence, starting at the specified position in the list.
boolean offer(E e)	It adds the specified element as the last element of a list.
boolean offerFirst(E e)	It inserts the specified element at the front of a list.
boolean offerLast(E e)	It inserts the specified element at the end of a list.
E peek()	It retrieves the first element of a list
E peekFirst()	It retrieves the first element of a list or returns null if a list is empty.
E peekLast()	It retrieves the last element of a list or returns null if a list is empty.
E poll()	It retrieves and removes the first element of a list.
E pollFirst()	It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast()	It retrieves and removes the last element of a list, or returns null if a list is empty.
E pop()	It pops an element from the stack represented by a list.

void push(E e)	It pushes an element onto the stack represented by a list.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
E removeFirst()	It removes and returns the first element from a list.
boolean removeFirstOccurrence(Object o)	It removes the first occurrence of the specified element in a list (when traversing the list from head to tail).
E removeLast()	It removes and returns the last element from a list.
boolean removeLastOccurrence(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
E set(int index, E element)	It replaces the element at the specified position in a list with the specified element.
int size()	It returns the number of elements in a list.

Add Elements to LinkedList

To add elements into LinkedList, we are using `add()` method. It adds elements into the list in the insertion order.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating LinkedList

    LinkedList< String> linkedList = new LinkedList< String>();

    linkedList.add("Delhi");

    linkedList.add("NewYork");

    linkedList.add("Moscow");

    linkedList.add("Dubai");

    // Displaying LinkedList

    System.out.println(linkedList);

  }

}
```

Copy

```
[Delhi, NewYork, Moscow, Dubai]
```

Removing Elements

To remove elements from the list, we are using remove method that remove the specified elements. We can also pass index value to remove the elements of it.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating LinkedList

    LinkedList< String> linkedList = new LinkedList< String>();

    linkedList.add("Delhi");
    linkedList.add("NewYork");

    linkedList.add("Moscow");

    linkedList.add("Dubai");

    // Displaying LinkedList

    System.out.println(linkedList);

    // Removing Elements

    linkedList.remove("Moscow");

    System.out.println("After Deleting Elements: \n"+linkedList);

    // Removing Second Element

    linkedList.remove(1);

    System.out.println("After Deleting Elements: \n"+linkedList);

  }

}
```

Copy

```
[Delhi, NewYork, Moscow, Dubai]

After Deleting Elements:

[Delhi, NewYork, Dubai]

After Deleting Elements:

[Delhi, Dubai]
```

Traversing Elements of LinkedList

Since LinkedList is a collection then we can use loop to iterate its elements. In this example we are traversing elements. See the below example.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating LinkedList

    LinkedList< String> linkedList = new LinkedList< String>();

    linkedList.add("Delhi");
    linkedList.add("NewYork");

    linkedList.add("Moscow");

    linkedList.add("Dubai");

    // Traversing ArrayList

    for(String element : linkedList) {

      System.out.println(element);

    }

  }

}
```

Copy

```
Delhi

NewYork

Moscow

Dubai
```

Get size of LinkedList

Sometimes we want to know number of elements an LinkedList holds. In that case we use size() then returns size of LinkedList which is equal to number of elements present in the list.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating LinkedList

    LinkedList< String> linkedList = new LinkedList< String>();

    linkedList.add("Delhi");

    linkedList.add("NewYork");

    linkedList.add("Moscow");

    linkedList.add("Dubai");

    // Traversing ArrayList
    for(String element : linkedList) {

      System.out.println(element);

    }

System.out.println("Total Elements: "+linkedList.size());

  }

}
```

Copy

```
Delhi

NewYork

Moscow

Dubai

Total Elements: 4
```

Sorting LinkedList Elements

To sort elements of an LinkedList, Java provides a class Collections that includes a static method sort(). In this example, we are using sort method to sort the elements.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

    // Creating LinkedList

    LinkedList< String> linkedList = new LinkedList< String>();

    linkedList.add("Delhi");

    linkedList.add("NewYork");

    linkedList.add("Moscow");

    linkedList.add("Dubai");

    // Sorting elements

    Collections.sort(linkedList);

    // Traversing ArrayList

    for(String element : linkedList) {
         System.out.println(element);

      }

   }

}
```

Copy

```
OUTPUT-

Delhi

Dubai

Moscow

NewYork
```

Java Collection Framework HashSet

Java HashSet class is used to store unique elements. It uses hash table internally to store the elements. It implements Set interface and extends the AbstractSet class. Declaration of the is given below.

Java HashSet class declaration

```
public class HashSet<E>extends AbstractSet<E>implements Set<E>, Cloneable, Serializable
```

Copy

Important Points:

1. It creates a collection that uses hash table for storage. A hash table stores information by using a mechanism called hashing.
2. HashSet does not maintain any order of elements.
3. HashSet contains only unique elements.
4. It allows to store null value.
5. It is non synchronized.
6. It is the best approach for search operations.
7. The initial default capacity of HashSet is 16.

HashSet Constructors

HashSet class has three constructors that can be used to create HashSet accordingly.

```
HashSet()  //This creates an empty HashSet



HashSet( Collection C )  //This creates a HashSet that is initialized with the elements of the Collection C



HashSet( int capacity )  //This creates a HashSet that has the specified initial capacity
```

Copy

Example of HashSet class

In this example, we are creating a HashSet which is initially empty. Later on we will add items to this collection.

```
import java.util.*;

class Demo

{
  public static void main(String args[])

  {

    // Creating HashSet

    HashSet<String> hs = new HashSet<String>();

    // Displaying HashSet

    System.out.println(hs);

  }

}
```

Copy

```
[]
```

HashSet Method

Method	Description
add(E e)	It adds the specified element to this set if it is not already present.
clear()	It removes all of the elements from the set.
clone()	It returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
contains(Object o)	It returns true if this set contains the specified element.
isEmpty()	It returns true if this set contains no elements.
iterator()	It returns an iterator over the elements in this set.
remove(Object o)	It removes the specified element from this set if it is present.
size()	It returns the number of elements in the set.

spliterator()	It creates a late-binding and fail-fast Spliterator over the elements in the set.

Add Elements to HashSet

In this example, we are creating a HashSet that store string values. Since HashSet does not store duplicate elements, we tried to add a duplicate elements but the output contains only unique elements.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

    // Creating HashSet

    HashSet<String> hs = new HashSet<String>();

    // Adding elements

    hs.add("Mohan");

    hs.add("Rohan");

    hs.add("Sohan");

    hs.add("Mohan");

    // Displaying HashSet

    System.out.println(hs);

  }

}
```

Copy

```
[Mohan, Sohan, Rohan]
```

Remove Elements from HashSet

To remove elements from the hashset, we are using remove() method that remove the specified elements.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

    // Creating HashSet

    HashSet<String> hs = new HashSet<String>();

    // Adding elements

    hs.add("Mohan");

    hs.add("Rohan");

    hs.add("Sohan");

    hs.add("Mohan");

    // Displaying HashSet

    System.out.println(hs);

    // Removing elements

    hs.remove("Mohan");

    System.out.println("After removing elements: \n"+hs);

  }

}
```

Copy

```
[Mohan, Sohan, Rohan]

After removing elements:

[Sohan, Rohan]
```

Traversing Elements of HashSet

Since HashSet is a collection then we can use loop to iterate its elements. In this example we are traversing elements using for loop. See the below example.

```
import java.util.*;

class Demo

{
  public static void main(String args[])

  {

    // Creating HashSet

    HashSet<String> hs = new HashSet<String>();

    // Adding elements

    hs.add("Mohan");

    hs.add("Rohan");

    hs.add("Sohan");

    hs.add("Mohan");

    // Traversing ArrayList

    for(String element : hs) {

      System.out.println(element);

    }

  }

}
```

Copy

```
Mohan

Sohan

Rohan
```

Get size of HashSet

Sometimes we want to know number of elements an HashSet holds. In that case we use size() then returns size of HashSet which is equal to number of elements present in the list.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

    // Creating HashSet

    HashSet<String> hs = new HashSet<String>();
    // Adding elements

    hs.add("Mohan");

    hs.add("Rohan");

    hs.add("Sohan");

    hs.add("Mohan");

    // Traversing ArrayList

    for(String element : hs) {

      System.out.println(element);

    }

    System.out.println("Total elements : "+hs.size());

  }

}
```

Copy

```
OUTPUT-

Mohan

Sohan

Rohan

Total elements : 3
```

Java Collection Framework Linked HashSet

Java LinkedHashSet class is a linked list based implementation of Set interface. It is used to store unique elements. It uses hash table internally to store the elements. It implements Set interface and extends the HashSet class. Declaration of the class is given below.

Java LinkedHashSet class declaration

```
public class LinkedHashSet<E>extends HashSet<E>implements Set<E>,
Cloneable, Serializable
```

Copy

Important Points:

1. LinkedHashSet class extends HashSet class
2. It maintains a linked list of entries in the set.
3. It contains unique elements only
4. It allows to insert null value.
5. It stores elements in the order in which elements are inserted i.e it maintains the insertion order.
6. Java LinkedHashSet class is non synchronized.

LinkedHashSet Constructors

LinkedHashSet class has four constructors that can be used to create LinkedHashSet accordingly.

```


LinkedHashSet()  // This creates an empty LinkedHashSet

LinkedHashSet( Collection C )  // This creates a LinkedHashSet that is
initialized with the elements of the Collection C

LinkedHashSet( int capacity )  // This creates a LinkedHashSet that has
the specified initial capacity

LinkedHashSet( int initialCapacity, float loadFactor )
```

Copy

Example of LinkedHashSet class

In this example, we are creating a LinkedHashSet which is initially empty. Later on we will add items to this collection.

```
import java.util.*;

import java.util.*;

class Demo
```

```
{
  public static void main(String args[])
  {
    // Creating LinkedHashSet
    LinkedHashSet<String> hs = new LinkedHashSet<String>();
    // Displaying LinkedHashSet
    System.out.println(hs);
  }
}
```
Copy

```
[]
```

LinkedHashSet Method

It Inherits methods from HashSet class. So we can apply all the HashSet operations with LinkedHashSet. The given table contains the methods of HashSet.

Method	Description
add(E e)	It adds the specified element to this set if it is not already present.
clear()	It removes all of the elements from the set.
clone()	It returns a shallow copy of this LinkedHashSet instance: the elements themselves are not cloned.
contains(Object o)	It returns true if this set contains the specified element.
isEmpty()	It returns true if this set contains no elements.
iterator()	It returns an iterator over the elements in this set.
remove(Object o)	It removes the specified element from this set if it is present.

size()	It returns the number of elements in the set.
spliterator()	It creates a late-binding and fail-fast Spliterator over the elements in the set.

Add Elements to LinkedHashSet

In this example, we are creating a LinkedHashSet that store integer values. Since LinkedHashSet does not store duplicate elements, we tried to add a duplicate elements but the output contains only unique elements.

```java
import java.util.*;

class Demo

{

  public static void main(String args[])

   {

     // Creating LinkedHashSet

     LinkedHashSet hs = new LinkedHashSet<>();

     // Adding elements

     hs.add(100);

     hs.add(200);

     hs.add(300);

     hs.add(100);

     // Displaying LinkedHashSet

     System.out.println(hs);

   }

}
```

Copy

```
[100, 200, 300]
```

Remove Elements from LinkedHashSet

To remove elements from the LinkedHashSet, we are using remove() method that remove the specified elements.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

    // Creating LinkedHashSet

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

    // Adding elements

    hs.add(100);

    hs.add(200);

    hs.add(300);

    hs.add(100);

    // Displaying LinkedHashSet

    System.out.println(hs);

    // Removing elements

    hs.remove(300);

    System.out.println("After removing elements: \n"+hs);

  }

}
```

Copy

```
[100, 200, 300] After removing elements: [100, 200]
```

Traversing Elements of LinkedHashSet

Since LinkedHashSet is a collection then we can use loop to iterate its elements. In this example we are traversing elements using for loop. See the below example.

```
import java.util.*;

class Demo

{

  public static void main(String args[])
  {

    // Creating LinkedHashSet

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

    // Adding elements

    hs.add(100);

    hs.add(200);

    hs.add(300);

    hs.add(100);

    // Traversing ArrayList

    for(Integer element : hs) {

      System.out.println(element);

    }

  }

}
```

Copy

```
100

200

300
```

Get size of LinkedHashSet

Sometimes we want to know number of elements an LinkedHashSet holds. In that case we use size() then returns size of LinkedHashSet which is equal to number of elements present in the list.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

    // Creating LinkedHashSet

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

    // Adding elements
    hs.add(100);

    hs.add(200);

    hs.add(300);

    hs.add(100);

    // Traversing ArrayList

    for(Integer element : hs) {

      System.out.println(element);

    }

    System.out.println("Total elements : "+hs.size());

  }

}
```

Copy

```
OUTPUT

100

200

300

Total elements : 3
```

Java Collection Framework Treeset

TreeSet class used to store unique elements in ascending order. It is similar to **HashSet** except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order.

Java TreeSet class implements the Set interface and use tree based data structure storage. It extends AbstractSet class and implements the NavigableSet interface. The declaration of the class is given below.

TreeSet class Declaration

```
public class TreeSet<E>extends AbstractSet<E>implements NavigableSet<E>, Cloneable, Serializable
```

Important Points

1. It stores the elements in ascending order.
2. It uses a Tree structure to store elements.
3. It contains unique elements only like HashSet.
4. It's access and retrieval times are quite fast.
5. It doesn't allow null element.
6. It is non synchronized.

TreeSet Constructors

```
TreeSet()

TreeSet( Collection C )

TreeSet( Comparator comp )

TreeSet( SortedSet ss )
```

Copy

Example: Creating an TreeSet

Lets create a TreeSet to store string elements. Here set is empty because we did not add elements to it.

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

   {
    // Creating an TreeSet

    TreeSet< String> fruits = new TreeSet< String>();


    // Displaying TreeSet

    System.out.println(fruits);

  }

}
```

Copy

```
[]
```

TreeSet Methods

Method	Description
boolean add(E e)	It adds the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It adds all of the elements in the specified collection to this set.
E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns comparator that arranged elements in order.
Iterator descendingIterator()	It is used iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified

	element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending order.
E lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.
E pollLast()	It is used to retrieve and remove the highest(last) element.
Spliterator spliterator()	It is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	It returns a set of elements that lie between the given range.
SortedSet subSet(E fromElement, E toElement))	It returns a set of elements that lie between the given range

	which includes fromElement and excludes toElement.
SortedSet tailSet(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns 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 clear()	It is used to remove all of the elements from this set.
Object clone()	It returns a shallow copy of this TreeSet instance.
E first()	It returns the first (lowest) element currently in this sorted set.
E last()	It returns the last (highest) element currently in this sorted set.
int size()	It returns the number of elements in this set.

Add Elements To TreeSet

Lets take an example to create a TreeSet that contains duplicate elements. But you can notice that it prints unique elements that means it does not allow duplicate elements.

```
import java.util.*;

class Demo{

  public static void main(String args[]){

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

    al.add("Ravi");

    al.add("Vijay");

    al.add("Ravi");

    al.add("Ajay");


    Iterator itr=al.iterator();

    while(itr.hasNext()){

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

    }

  }

}
```

Copy

```
Ajay

Ravi

Vijay
```

Removing Elements From the TreeSet

We can use `remove()` method of this class to remove the elements. See the below example.

```
import java.util.*;

class Demo{

  public static void main(String args[]){

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

    al.add("Ravi");

    al.add("Vijay");

    al.add("Ravi");

    al.add("Ajay");
    Iterator itr=al.iterator();
    while(itr.hasNext()){
      System.out.println(itr.next());
    }

    al.remove("Ravi");
    System.out.println("After Removing: "+al);
  }
}
```

Copy

```
Ajay
Ravi
Vijay
After Removing: [Ajay, Vijay]
```

Search an Element in TreeSet

TreeSet provides `contains()` method that true if elements is present in the set.

```
import java.util.*;
class Demo{
  public static void main(String args[]){
    TreeSet<String> al = new TreeSet<String>();
    al.add("Ravi");
    al.add("Vijay");
    al.add("Ravi");
    al.add("Ajay");

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


    boolean iscontain = al.contains("Ravi");

    System.out.println("Is contain Ravi: "+iscontain);

  }

}
```

Copy

```
Ajay

Ravi

Vijay

Is contain Ravi: true
```

Traverse TreeSet in Ascending and Descending Order

We can traverse elements of Treeset in both ascending and descending order. TreeSet provides descendingIterator() method that returns iterator type for descending traversing. See the below example.

```
import java.util.*;

class Demo{

  public static void main(String args[]){

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

    al.add("Ravi");

    al.add("Vijay");

    al.add("Ravi");

    al.add("Ajay");

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

    Iterator itr=al.iterator();

    while(itr.hasNext()){

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

    }

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

    Iterator itr2=al.descendingIterator();
    while(itr2.hasNext()){

      System.out.println(itr2.next());

    }

  }

}
```

Copy

```
OUTPUT

Ascending...

Ajay

Ravi

Vijay

Descending...

Vijay

Ravi

Ajay
```

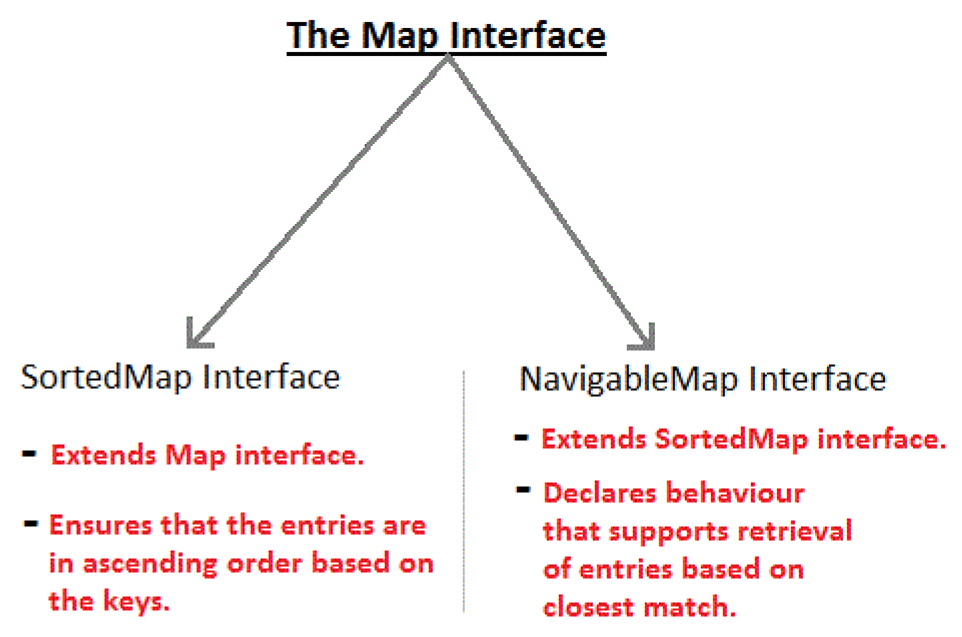
Map Interface - Java Collections

A Map stores data in key and value association. Both key and values are objects. The key must be unique but the values can be duplicate. Although Maps are a part of Collection Framework, they can not actually be called as collections because of some properties that they posses. However we can obtain a **collection-view** of maps.

It provides various classes: **HashMap, TreeMap, LinkedHashMap** for map implementation. All these classes implements Map interface to provide Map properties to the collection.

Map Interface and its Subinterface

Interface	Description
Map	Maps unique key to value.
Map.Entry	Describe an element in key and value pair in a map. Entry is sub interface of Map.
NavigableMap	Extends SortedMap to handle the retrienal of entries based on closest match searches
SortedMap	Extends Map so that key are maintained in an ascending order.


The Map Interface
SortedMap Interface
- Extends Map interface.
- Ensures that the entries are in ascending order based on the keys.
NavigableMap Interface
- Extends SortedMap interface.
- Declares behaviour that supports retrieval of entries based on closest match.

Map Interface Methods

These are commonly used methods defined by Map interface

- boolean **containsKey**(Object *k*): returns true if map contain *k* as key. Otherwise false.
- Object **get**(Object *k*) : returns values associated with the key *k*.
- Object **put**(Object *k*, Object *v*) : stores an entry in map.
- Object **putAll**(Map *m*) : put all entries from *m* in this map.

- Set **keySet**() : returns **Set** that contains the key in a map.
- Set **entrySet**() : returns **Set** that contains the entries in a map.

HashMap class

1. HashMap class extends **AbstractMap** and implements **Map** interface.
2. It uses a **hashtable** to store the map. This allows the execution time of `get()` and `put()` to remain same.
3. HashMap does not maintain order of its element.

HashMap has four constructor.

```
HashMap()

HashMap(Map< ? extends k, ? extends V> m)

HashMap(int capacity)

HashMap(int capacity, float fillratio)
```
Copy

HashMap Example

Lets take an example to create hashmap and store values in key and value pair. Notice to insert elements, we used `put()` method because map uses put to insert element, not `add()` method that we used in list interface.

```
import java.util.*;


class Demo

{

  public static void main(String args[])

  {

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

    hm.put("a",100);

    hm.put("b",200);

    hm.put("c",300);

    hm.put("d",400);



    Set<Map.Entry<String,Integer>> st = hm.entrySet();   //returns Set
view

    for(Map.Entry<String,Integer> me:st)
    {
      System.out.print(me.getKey()+":");
      System.out.println(me.getValue());
    }
  }
}
```

Copy

```
a:100
b:200
c:300
d:400
```

TreeMap class

1. TreeMap class extends **AbstractMap** and implements **NavigableMap** interface.
2. It creates Map, stored in a tree structure.
3. A **TreeMap** provides an efficient means of storing key/value pair in efficient order.
4. It provides key/value pairs in sorted order and allows rapid retrieval.

Example:

In this example, we are creating treemap to store data. It uses tree to store data and data is always in sorted order. See the below example.

```
import java.util.*;

class Demo
{
  public static void main(String args[])
  {
    TreeMap<String,Integer> tm = new TreeMap<String,Integer>();
    tm.put("a",100);
    tm.put("b",200);
    tm.put("c",300);
    tm.put("d",400);

    Set<Map.Entry<String,Integer>> st = tm.entrySet();
    for(Map.Entry<String,Integer> me:st)

    {

      System.out.print(me.getKey()+":");

      System.out.println(me.getValue());

    }

  }

}
```

Copy

```
a:100

b:200

c:300

d:400
```

LinkedHashMap class

1. **LinkedHashMap** extends **HashMap** class.
2. It maintains a linked list of entries in map in order in which they are inserted.
3. **LinkedHashMap** defines the following constructor

```
4. LinkedHashMap()

5.

6. LinkedHashMap(Map< ? extends k, ? extends V> m)

7.

8. LinkedHashMap(int capacity)

9.

10.LinkedHashMap(int capacity, float fillratio)

11.
```

```
LinkedHashMap(int capacity, float fillratio, boolean order)
```

Copy

12. It adds one new method removeEldestEntry(). This method is called by put() and putAll() By default this method does nothing.

Example:

Here we are using linkedhashmap to store data. It stores data into insertion order and use linked-list internally. See the below example.

```
import java.util.*;



class Demo

{

  public static void main(String args[])

  {

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

    tm.put("a",100);

    tm.put("b",200);

    tm.put("c",300);

    tm.put("d",400);



    Set<Map.Entry<String,Integer>> st = tm.entrySet();

    for(Map.Entry<String,Integer> me:st)

    {

      System.out.print(me.getKey()+":");

      System.out.println(me.getValue());

    }

  }

}
```

Copy

```
OUTPUT

a:100

b:200

c:300

d:400
```

Java Collection Framework Hashmap

Java HashMap class is an implementation of Map interface based on hash table. It stores elements in key & value pairs which is denoted as HashMap<Key, Value> or HashMap<K, V>.

It extends AbstractMap class, and implements Map interface and can be accessed by importing **java.util** package. Declaration of this class is given below.

HashMap Declaration

```
public class HashMap<K,V>extends AbstractMap<K,V>implements
Map<K,V>,Cloneable, Serializable
```

Copy

Important Points:

- It is member of the Java Collection Framework.
- It uses a hashtable to store the map. This allows the execution time of `get()` and `put()` to remain same.
- HashMap does not maintain order of its element.
- It contains values based on the key.
- It allows only unique keys.
- It is unsynchronized.
- Its initial default capacity is 16.
- It permits null values and the null key
- It is unsynchronized.

HashMap Constructors

HashMap class provides following four constructors.

```
HashMap()

HashMap(Map< ? extends k, ? extends V> m)

HashMap(int capacity)

HashMap(int capacity, float loadfactor)
```

Copy

Example: Creating a HashMap

Lets take an example to create a hashmap that can store integer type key and string values. Initially it is empty because we did not add elements to it. Its elements are enclosed into curly braces.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

  // Creating HashMap

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


    // Displaying HashMap

    System.out.println(hashMap);

  }

}
```

Copy

```
{}
```

Adding Elements To HashMap

After creating a hashmap, now lets add elements to it. HashMap provides `put()` method that takes two arguments first is key and second is value. See the below example.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

  // Creating HashMap

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

    // Adding elements

    hashMap.put(1, "One");

    hashMap.put(2, "Two");
```

```java
    hashMap.put(3, "Three");

    hashMap.put(4, "Four");

    // Displaying HashMap

    System.out.println(hashMap);

  }

}
```

Copy

```
{1=One, 2=Two, 3=Three, 4=Four}
```

Removing Elements From HashMap

In case, we need to remove any element from the hashmap. We can use remove() method that takes key as an argument. it has one overloaded remove() method that takes two arguments first is key and second is value.

```java
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

  // Creating HashMap

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

    // Adding elements

    hashMap.put(1, "One");

    hashMap.put(2, "Two");

    hashMap.put(3, "Three");

    hashMap.put(4, "Four");

    // Displaying HashMap

    System.out.println(hashMap);

    // Remove element by key

    hashMap.remove(2);

    System.out.println("After Removing 2 :\n"+hashMap);
    // Remove by key and value

    hashMap.remove(3, "Three");

    System.out.println("After Removing 3 :\n"+hashMap);


  }

}
```

Copy

```
{1=One, 2=Two, 3=Three, 4=Four}

After Removing 2 :

{1=One, 3=Three, 4=Four}

After Removing 3 :

{1=One, 4=Four}
```

Traversing Elements

To access elements of the hashmap, we can traverse them using the loop. In this example, we are using for loop to iterate the elements.

```
  import java.util.*;

class Demo

{

  public static void main(String args[])

  {

  // Creating HashMap

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

    // Adding elements

    hashMap.put(1, "One");

    hashMap.put(2, "Two");

    hashMap.put(3, "Three");

    hashMap.put(4, "Four");

    // Traversing HashMap

    for(Map.Entry<Integer, String> entry : hashMap.entrySet()) {

      System.out.println(entry.getKey()+" : "+entry.getValue());
    }

  }

}
```

Copy

```
1 : One

2 : Two

3 : Three

4 : Four
```

Replace HashMap Elements

HashMap provides built-in methods to replace elements. There are two overloaded replace methods: first takes two arguments one for key and second for the value we want to replace with. Second method takes three arguments first is key and second is value associated with the key and third is value that we want to replace with the key-value.

```
import java.util.*;

class Demo

{

  public static void main(String args[])

  {

  // Creating HashMap

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

    // Adding Elements

    hashMap.put(1, "One");

    hashMap.put(2, "Two");

    hashMap.put(3, "Three");

    hashMap.put(4, "Four");

    // Traversing HashMap

    for(Map.Entry<Integer, String> entry : hashMap.entrySet()) {

      System.out.println(entry.getKey()+" : "+entry.getValue());

    }

    // Replacing Elements of HashMap
    hashMap.replace(1, "One-1");

    System.out.println(hashMap);

    hashMap.replace(1, "One-1", "First");

    System.out.println(hashMap);

  }

}
```

Copy

```
OUTPUT

1 : One

2 : Two

3 : Three

4 : Four

{1=One-1, 2=Two, 3=Three, 4=Four}

{1=First, 2=Two, 3=Three, 4=Four}
```

Java Collection Classes

Java Collections class is a part of collection framework. This class is designed to provide methods for searching, sorting, copying etc. It consists exclusively of built-in static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections.

This class is located into **java.util** package. The declaration of this class is given below.

Collections Class Declaration

```
public class Collections extends Object
```

Copy

It inherits Object class and all the methods of this class throw a **NullPointerException** if the object is null.

Collections Methods

This table contains commonly used methods of Collections class.

Method	Description
addAll()	It adds all of the specified elements to the specified collection.
binarySearch()	It searches the list for the specified object and returns their position in a sorted list.
copy()	It copies all the elements from one list into another list.
disjoint()	It returns true if the two specified collections have no elements in common.
emptyEnumeration()	It fetches an enumeration that has no elements.
emptyIterator()	It fetches an Iterator that has no elements.
emptyList()	It fetches a List that has no elements.

emptyListIterator()	It fetches a List Iterator that has no elements.
emptyMap()	It returns an empty map which is immutable.
emptyNavigableMap()	It returns an empty navigable map which is immutable.
emptyNavigableSet()	It returns an empty navigable set which is immutable in nature.
emptySet()	It returns the set that has no elements.
emptySortedMap()	It returns an empty sorted map which is immutable.
emptySortedSet()	It is used to get the sorted set that has no elements.
enumeration()	It is used to get the enumeration over the specified collection.
fill()	It is used to replace all of the elements of the specified list with the specified elements.
list()	It is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration.
max()	It is used to get the maximum value of the given collection.

min()	It is used to get the minimum value of the given collection.
nCopies()	It is used to get an immutable list consisting of n copies of the specified object.
replaceAll()	It is used to replace all occurrences of one specified value in a list with the other specified value.
reverse()	It is used to reverse the order of the elements in the specified list.
reverseOrder()	It is used to get the comparator that imposes the reverse of the natural ordering on a collection.
rotate()	It is used to rotate the elements in the specified list by a given distance.
shuffle()	It is used to randomly reorders the specified list elements using a default randomness.
sort()	It is used to sort the elements presents in the specified list of collection in ascending order.
swap()	It is used to swap the elements at the specified positions in the specified list.
synchronizedCollection()	It is used to get a synchronized (thread-safe) collection backed by the specified collection.

synchronizedList()	It is used to get a synchronized (thread-safe) collection backed by the specified list.
synchronizedMap()	It is used to get a synchronized (thread-safe) map backed by the specified map.
synchronizedNavigableMap()	It is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map.
synchronizedNavigableSet()	It is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set.
synchronizedSet()	It is used to get a synchronized (thread-safe) set backed by the specified set.
synchronizedSortedMap()	It is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map.
synchronizedSortedSet()	It is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set.

Example: Sorting List

In this example, we are using sort() method of Collections class that is used to sort elements of a collection. Here we are using Arralist class that stores integer type object and sorted.

```
import java.util.*;

public class Demo {

    public static void main(String a[]){

      // Creating ArrayList

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

        // Adding elements
        list.add(100);

        list.add(2);

        list.add(66);

        list.add(22);

        list.add(10);

        // Displaying list

        System.out.println(list);

        // Sorting list

        Collections.sort(list);

        // Displaying sort data

        System.out.println(list);

    }

}
```

Copy

```
[100, 2, 66, 22, 10]

[2, 10, 22, 66, 100]
```

Example: Finding min and max elements

The collections class provides two methods `max()` and `min()` that can be used to fetch max and min values from a collection. See the below example.

```
import java.util.*;

public class Demo {

    public static void main(String a[]){

      // Creating ArrayList

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

        // Adding elements

        list.add(100);

        list.add(2);

        list.add(66);

        list.add(22);
        list.add(10);

        // Displaying list

        System.out.println(list);

        // Find min element

        int min = Collections.min(list);

        // Find max element

        int max = Collections.max(list);

        // Displaying data

        System.out.println("Minimum element : "+ min);

        System.out.println("Maximum element : "+ max);

    }

}
```

Copy

```
[100, 2, 66, 22, 10]

Minimum element : 2

Maximum element : 100
```

Example: Swapping Elements

To swap elements, we don't need write logic code. Collections class provides built-in swap method that can be used to swap elements from one position to another in a collection. The swap() method takes three arguments: first is reference of object, second is index of first elements and third is index of second elements to be swapped. See the below example.

```
import java.util.*;

public class Demo {

    public static void main(String a[]){

      // Creating ArrayList

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

        // Adding elements

        list.add(100);

        list.add(2);

        list.add(66);
        list.add(22);

        list.add(10);

        // Displaying list

        System.out.println(list);

        // Swapping elements

        Collections.swap(list, 0, 4); // 100 is swapped by 10

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

    }

}
```

Copy

```
[100, 2, 66, 22, 10]

List after swapping : [10, 2, 66, 22, 100]
```

Example: Reverse the list

Collections class provides a static method reverse() that is used to get a collection in reverse order. In the below example, we are getting list in reverse order using the reverse() method.

```
import java.util.*;

public class Demo {

    public static void main(String a[]){

      // Creating ArrayList

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

        // Adding elements

        list.add(100);

        list.add(2);

        list.add(66);

        list.add(22);

        list.add(10);

        // Displaying list

        System.out.println(list);

        // Reverse the list
        Collections.reverse(list);

        // Displaying data

        System.out.println("List in reverse order "+list);

    }

}
```

Copy

```
OUTPUT

[100, 2, 66, 22, 10]

List in reverse order [10, 22, 66, 2, 100]
```

Java Comparable Interface

Java Comparable interface is a member of collection framework which is used to compare objects and sort them according to the natural order.

The natural ordering refers to the behavior of `compareTo()` method which is defined into Comparable interface. Its sorting technique depends on the type of object used by the interface. If object type is string then it sorts it Lexicographically.

If object type is wrapper class object like: integer or list then it sorts according to their values.

If object type is custom object like: user defined object then sorts according to the defined `compareTo()` method.

Classes that implements this interface can be sorted automatically by calling `Collections.sort()` method. Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comaprator. Declaration of this interface is given below.

Declaration

```
public interface Comparable<T>
```

Copy

Comparable Method

It contains single method `compareTo()` that is given below.

int compareTo(T o) : It compares object with the specified object for order.

The `compareTo()` function compares the current object with the provided object. This function is already implemented for default wrapper classes and primitive data types but, this function also needs to be implemented for user-defined classes.

It **returns** positive integer, if the current object is greater than the provided object.

If the current object is less than the provided object then it returns negative integer.

If the current object is equal to the provided object then it returns zero.

Exceptions

This method returns NullPointerException, if the specified object is null and `ClassCastException` if the specified object's type prevents it from being compared to this object.

Example : Sorting list

Lets take an example to sort an ArrayList that stores integer values. We are using `sort()` method of Collections class that sort those object which implements Compares interface. Since integer wrapper class implements Comparable so we are able to get sorted objects. See the below example.

```
import java.util.*;

public class Demo {
    public static void main(String a[]){

      // Creating List

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

        // Adding elements

        list.add(100);

        list.add(2);

        list.add(66);

        list.add(22);

        list.add(10);

        // Displaying list

        System.out.println(list);

        // Sorting list

        Collections.sort(list);

        // Displaying sorted list

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

    }

}
```

Copy

```
[100, 2, 66, 22, 10]
Sorted List : [2, 10, 22, 66, 100]
```

Example: Sorting String objects

While sorting string objects, the comparable sorts it based on lexicographically. It means a dictionary like sorting order. See the below example.

```
import java.util.*;

public class Demo {

    public static void main(String a[]){

      // Creating List

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

        // Adding elements
        list.add("D");

        list.add("L");

        list.add("A");

        list.add("Z");

        list.add("C");

        // Displaying list

        System.out.println(list);

        // Sorting list

        Collections.sort(list);

        // Displaying sorted list

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

    }
```

Copy

```
[D, L, A, Z, C]

Sorted List : [A, C, D, L, Z]
```

Example: Sorting User Defined Object

If we have custom objects then we have to implement the Comparable interface and override its compareTo() method. Now it will compare based on the logic we defined in our `compareTo()` method. See the below example.

```
import java.util.*;


class Employee implements Comparable <Employee>

{

    int empId;

    String name;

    public Employee (int empId, String name)

    {

        this.empId=empId;

        this.name=name;
    }

    public String toString()

    {

        return this.empId + " " + this.name;

    }


     // Sorting by empId

    public int compareTo(Employee std){


        return this.empId - std.empId;

    }

}


public class Demo {

    public static void main(String a[]){

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

        list.add(new Employee(2, "Boman"));

        list.add(new Employee(1, "Abram"));

        list.add(new Employee(3, "Dinesh"));


        // Displaying

        for (int i=0; i<list.size(); i++)

            System.out.println(list.get(i));


        // Sorting

        Collections.sort(list);


        // Displaying after sorting

        System.out.println("\nAfter Sorting :\n");

        for (int i=0; i<list.size(); i++)
```

```
            System.out.println(list.get(i));

    }

}
```

Copy

```
OUTPUT

2 Boman

1 Abram

3 Dinesh


After Sorting :


1 Abram

2 Boman

3 Dinesh
```

Comparator Interface - Java Collections

In Java, Comparator interface is used to order(sort) the objects in the collection in your own way. It gives you the ability to decide how elements will be sorted and stored within collection and map.

Comparator Interface defines `compare()` method. This method has two parameters. This method compares the two objects passed in the parameter. It returns 0 if two objects are equal. It returns a positive value if object1 is greater than object2. Otherwise a negative value is returned. The method can throw a **ClassCastException** if the type of object are not compatible for comparison.

Rules for using Comparator interface

Rules for using Comparator interface:

1. If you want to sort the elements of a collection, you need to implement Comparator interface.
2. If you do not specify the type of the object in your Comparator interface, then, by default, it assumes that you are going to sort the objects of type Object. Thus, when you override the compare() method ,you will need to specify the type of the parameter as Object only.
3. If you want to sort the user-defined type elements, then while implementing the Comparator interface, you need to specify the user-defined type generically. If you do not specify the user-defined type while implementing the interface,then by default, it assumes Object type and you will not be able to compare the user-defined type elements in the collection

For Example:

If you want to sort the elements according to roll number, defined inside the class Student, then while implementing the Comparator interface, you need to mention it generically as follows:

```
class MyComparator implements Comparator<Student>{}
```

Copy
If you write only,

```
class MyComparator implements Comparator {}
```

Copy
Then it assumes, by default, data type of the compare() method's parameter to be Object, and hence you will not be able to compare the Student type(user-defined type) objects.

Time for an Example!

Student class:

```
class Student

int roll;

  String name;
  Student(int r,String n)

  {

      roll = r;

      name = n;

  }

  public String toString()

  {

      return roll+" "+name;

  }
```

Copy

MyComparator class:

This class defines the comparison logic for Student class based on their roll. Student object will be sorted in ascending order of their roll.

```
class MyComparator implements Comparator<Student>

{

  public int compare(Student s1,Student s2)

    {

        if(s1.roll == s2.roll) return 0;

        else if(s1.roll > s2.roll) return 1;

        else return -1;

    }

}
```

Copy

Now let's create a Test class with `main()` function,

```
public class Test

{



   public static void main(String[] args)

    {

       TreeSet< Student> ts = new TreeSet< Student>(new
MyComparator());

       ts.add(new Student(45, "Rahul"));
```

```
        ts.add(new Student(11, "Adam"));

        ts.add(new Student(19, "Alex"));

        System.out.println(ts);

    }



}
```

Copy

```
[ 11 Adam, 19 Alex, 45 Rahul ]
```

As you can see in the ouput Student object are stored in ascending order of their **roll**.

Note:

- When we are sorting elements in a collection using Comparator interface, we need to pass the class object that implements Comparator interface.
- To sort a TreeSet collection, this object needs to be passed in the constructor of TreeSet.
- If any other collection, like ArrayList,was used, then we need to call sort method of Collections class and pass the name of the collection and this object as a parameter.
- For example, If the above program used ArrayList collection, the public class test would be as follows:

```
public class Test

{

    public static void main(String[] args)

    {

        ArrayList< Student> ts = new ArrayList< Student>();

        ts.add(new Student(45, "Rahul"));

        ts.add(new Student(11, "Adam"));

        ts.add(new Student(19, "Alex"));

        Collections.sort(ts,new MyComparator()); /*passing the name of
the ArrayList and the

object of the class that implements Comparator in a predefined sort()
method in Collections

class*/

        System.out.println(ts);

    }

}
```

Legacy Classes - Java Collections

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.
```

Copy

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

```
Vector() //This creates a default vector, which has an initial size of 10.

Vector(int size) //This creates a vector whose initial capacity is specified by size.

Vector(int size, int incr) //This creates a vector whose initial capacity is specified by size and whose
increment is specified by incr. The increment specifies the number of elements to allocate each time
when a vector is resized for addition of objects.

```

```
Vector(Collection c) //This creates a vector that contains the elements of collection c.
```

Copy

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<Integer> ve = new Vector<Integer>();

         ve.add(10);

         ve.add(20);

         ve.add(30);

         ve.add(40);

         ve.add(50);

         ve.add(60);


         Enumeration<Integer> en = ve.elements();


         while(en.hasMoreElements())

         {

             System.out.println(en.nextElement());

         }

    }



}
```

Copy

```
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.

14.Hashtable(Map< ? extends K, ? extends V> m) //This creates a hash table that is initialized with the

15.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.
```

Copy

Example of Hashtable

```java
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());

    }

  }

}
```

Copy

```
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.
```

Copy

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) );

      }



    }



}
```

Copy

```
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()+" ");


}
}
```

Copy

```
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.
