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
```