Collections

The Collection framework

The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table –

Sr.No.	Class & Description
1	**AbstractCollection** Implements most of the Collection interface.
2	**AbstractList** Extends AbstractCollection and implements most of the List interface.
3	**AbstractSequentialList** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4	LinkedList Implements a linked list by extending AbstractSequentialList.
5	ArrayList Implements a dynamic array by extending AbstractList.
6	**AbstractSet** Extends AbstractCollection and implements most of the Set interface.

7	HashSet Extends AbstractSet for use with a hash table.
8	LinkedHashSet Extends HashSet to allow insertion-order iterations.
9	TreeSet Implements a set stored in a tree. Extends AbstractSet.
10	**AbstractMap** Implements most of the Map interface.
11	HashMap Extends AbstractMap to use a hash table.
12	TreeMap Extends AbstractMap to use a tree.
13	WeakHashMap Extends AbstractMap to use a hash table with weak keys.
14	LinkedHashMap Extends HashMap to allow insertion-order iterations.
15	IdentityHashMap Extends AbstractMap and uses reference equality when comparing documents.

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

LinkedList

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

Following are the constructors supported by the LinkedList class.

Sr.No.	Constructor & Description

1	**LinkedList( )** This constructor builds an empty linked list.
2	**LnkedList(Collection c)** This constructor builds a linked list that is initialized with the elements of the collection **c**.

Apart from the methods inherited from its parent classes, LinkedList defines following methods –

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**boolean addAll(Collection c)** 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. Throws NullPointerException if the specified collection is null.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
5	**void addFirst(Object o)** Inserts the given element at the beginning of this list.
6	**void addLast(Object o)** Appends the given element to the end of this list.

7	**void clear()** Removes all of the elements from this list.
8	**Object clone()** Returns a shallow copy of this LinkedList.
9	**boolean contains(Object o)** Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).
10	**Object get(int index)** Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
11	**Object getFirst()** Returns the first element in this list. Throws NoSuchElementException if this list is empty.
12	**Object getLast()** Returns the last element in this list. Throws NoSuchElementException if this list is empty.
13	**int indexOf(Object o)** Returns the index in this list of the first occurrence of the specified element, or -1 if the list does not contain this element.
14	**int lastIndexOf(Object o)** Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
15	**ListIterator listIterator(int index)** Returns a list-iterator of the elements in this list (in proper sequence), starting at the

	specified position in the list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
16	**Object remove(int index)** Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.
17	**boolean remove(Object o)** Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
18	**Object removeFirst()** Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.
19	**Object removeLast()** Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.
20	**Object set(int index, Object element)** Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).
21	**int size()** Returns the number of elements in this list.
22	**Object[] toArray()** Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
23	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this list in the correct order; the

	runtime type of the returned array is that of the specified array.

Example

The following program illustrates several of the methods supported by LinkedList:

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

   public static void main(String args[]) {
      // create a linked list
      LinkedList ll = new LinkedList();

      // add elements to the linked list
      ll.add("F");
      ll.add("B");
      ll.add("D");
      ll.add("E");
      ll.add("C");
      ll.addLast("Z");
      ll.addFirst("A");
      ll.add(1, "A2");
      System.out.println("Original contents of ll: " + ll);

      // remove elements from the linked list
      ll.remove("F");
      ll.remove(2);
      System.out.println("Contents of ll after deletion: " + ll);

      // remove first and last elements
      ll.removeFirst();
      ll.removeLast();
      System.out.println("ll after deleting first and last: " + ll);

      // get and set a value
      Object val = ll.get(2);
```

```
      ll.set(2, (String) val + " Changed");
      System.out.println("ll after change: " + ll);
   }
}
```

This will produce the following result –

Output

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

ArrayList

The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Following is the list of the constructors provided by the ArrayList class.

Sr.No.	Constructor & Description
1	**ArrayList( )** This constructor builds an empty array list.
2	**ArrayList(Collection c)** This constructor builds an array list that is initialized with the elements of the collection **c**.
3	**ArrayList(int capacity)** This constructor builds an array list that has the specified initial capacity. The capacity

	is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

Apart from the methods inherited from its parent classes, ArrayList defines the following methods –

Sr.No.	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index > size()).
2	**boolean add(Object o)** Appends the specified element to the end of this list.
3	**boolean addAll(Collection c)** 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. Throws NullPointerException, if the specified collection is null.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
5	**void clear()** Removes all of the elements from this list.
6	**Object clone()** Returns a shallow copy of this ArrayList.
7	**boolean contains(Object o)** Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element **e** such that (o==null ? e==null : o.equals(e)).

void ensureCapacity(int minCapacity)

8 Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

Object get(int index)

9 Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).

int indexOf(Object o)

10 Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

int lastIndexOf(Object o)

11 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 remove(int index)

12 Removes the element at the specified position in this list. Throws IndexOutOfBoundsException if the index out is of range (index < 0 || index >= size()).

protected void removeRange(int fromIndex, int toIndex)

13 Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.

Object set(int index, Object element)

14 Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).

int size()

15 Returns the number of elements in this list.

16 **Object[] toArray()**

Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.

Object[] toArray(Object[] a)

17 Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

void trimToSize()

18 Trims the capacity of this ArrayList instance to be the list's current size.

Example

The following program illustrates several of the methods supported by ArrayList –

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

   public static void main(String args[]) {
      // create an array list
      ArrayList al = new ArrayList();
      System.out.println("Initial size of al: " + al.size());

      // add elements to the array list
      al.add("C");
      al.add("A");
      al.add("E");
      al.add("B");
      al.add("D");
      al.add("F");
      al.add(1, "A2");
      System.out.println("Size of al after additions: " + al.size());

      // display the array list
      System.out.println("Contents of al: " + al);

      // Remove elements from the array list
      al.remove("F");
```

```
      al.remove(2);
      System.out.println("Size of al after deletions: " + al.size());
      System.out.println("Contents of al: " + al);
   }
}
```

This will produce the following result –

Output

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

HashMap

The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.

Following is the list of constructors supported by the HashMap class.

Sr.No.	Constructor & Description
1	**HashMap( )** This constructor constructs a default HashMap.
2	**HashMap(Map m)** This constructor initializes the hash map by using the elements of the given Map object **m**.
3	**HashMap(int capacity)** This constructor initializes the capacity of the hash map to the given integer value, capacity.

4	**HashMap(int capacity, float fillRatio)** This constructor initializes both the capacity and fill ratio of the hash map by using its arguments.

Apart from the methods inherited from its parent classes, HashMap defines the following methods –

Sr.No.	Method & Description
1	**void clear()** Removes all mappings from this map.
2	**Object clone()** Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
3	**boolean containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
4	**boolean containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
5	**Set entrySet()** Returns a collection view of the mappings contained in this map.
6	**Object get(Object key)** Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
7	**boolean isEmpty()** Returns true if this map contains no key-value mappings.
8	**Set keySet()** Returns a set view of the keys contained in this map.

9	**Object put(Object key, Object value)** Associates the specified value with the specified key in this map.
10	**putAll(Map m)** Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
11	**Object remove(Object key)** Removes the mapping for this key from this map if present.
12	**int size()** Returns the number of key-value mappings in this map.
13	**Collection values()** Returns a collection view of the values contained in this map.

Example

The following program illustrates several of the methods supported by this collection –

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

   public static void main(String args[]) {

      // Create a hash map
      HashMap hm = new HashMap();

      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
      hm.put("Qadir", new Double(-19.08));
```

```
      // Get a set of the entries
      Set set = hm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)hm.get("Zara")).doubleValue();
      hm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + hm.get("Zara"));
   }
}
```

This will produce the following result –

Output

```
Daisy: 99.22
Ayan: 1378.0
Zara: 3434.34
Qadir: -19.08
Mahnaz: 123.22

Zara's new balance: 4434.34
```

The Collection Algorithms

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

Sr.No.	Algorithm & Description
1	The Collection Algorithms Here is a list of all the algorithm implementation.

How to Use an Iterator ?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

Sr.No.	Iterator Method & Description
1	Using Java Iterator Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces.

How to Use a Comparator ?

Both TreeSet and TreeMap store elements in a sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify).

Sr.No.	Iterator Method & Description
1	Using Java Comparator Here is a list of all the methods with examples provided by Comparator Interface.

Summary

The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.

A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.

The classes and interfaces of the collections framework are in package java.util.