Chapter - 13

Collections

Collections

Introduction

Term "Collections". The overall term for Java's data structure facilities is ***Collections***, a term is an alternative to the more common ***Data Structures***. In addition to describing general data structure facilities, ***java.util.Collection*** is the name of an interface and ***java.util.Collections*** is the name of a class containing many data structure utility methods. *Three different names would hve been better.*

Use the Java library - Don't write your own

Most data structure textbooks give students the impression that professional programmers implement basic data structures, such as linked lists, or write their own sort methods.

This is completely false! Many standard data structures and algorithms have already been written and are available in Java. Professional programmers use these library classes as building blocks for their application specific structures.

Textbooks give this distorted view because their emphasis is on *understanding* how data structures are build using only pointers/references and arrays. It's an excellent exercise and is necessary to becoming a good programmer. But textbooks often fail to emphasize the predefined data structure libraries that programmers actually use.

You will, of course, often write data structures that reflect the nature of your problem, but use the library versions for the basic elements. And you will have plenty of opportunities to use this basic knowledge to beyond the basics.

To be a productive programmer, your first choice must be to use standard library data structures. In Java this means using the **Collection** classes and interfaces.

You will always have to write some of your own data structures, but you can program faster, and produce more robust and readable programs by building on the work of others.

No primitive types. **Collections** data structures work only with objects, not primitive values. You can use the wrapper classes (somewhat more convenient in Java 5), use alternate libraries, or write your own class equivalents. The creation and garbage collection of these extra objects may

add substantial overhead. There are several non-Sun implementations of data structures using primitives.

A ***collection*** — sometimes called a container — is simply an object that groups multiple elements into a single unit. **Collections** are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

What Is a Collections Framework?

A ***collections framework*** is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

Interfaces: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

Implementations: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

Reduces programming effort: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated

APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

Increases program speed and quality: This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

Allows interoperability among unrelated APIs: The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

Reduces effort to learn and to use new APIs: Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.

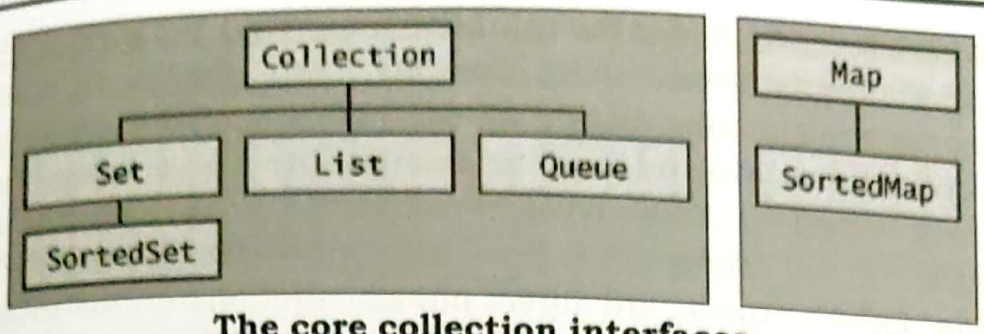
Reduces effort to design new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

Fosters software reuse: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework.

As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A Set is a special kind of **Collection**, a **SortedSet** is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees — a **Map** is not a true **Collection**.

Note that all the core collection interfaces are generic. For example, this is the declaration of the **Collection** interface.

public interface Collection<E>...

The <E> syntax tells you that the interface is generic. When you declare a **Collection** instance you can and should specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework. This chapter discusses general guidelines for effective use of the interfaces, including when to use which interface. You'll also learn programming idioms for each interface to help you get the most out of it.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated *optional* — a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an **UnsupportedOperationException**. Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.

List — an ordered collection (sometimes called *a sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.

Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a **FIFO** (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a **FIFO** queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties. **Map** — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used **Hashtable**, you're already familiar with the basics of Map.

The last two core collection interfaces are merely sorted versions of Set and **Map**:

SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of **SortedSet**. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

To understand how the sorted interfaces maintain the order of their elements.

The Collection Interface

A **Collection** represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a **Collection<String> c**, which may be a List, a Set, or another kind of **Collection**. This idiom creates a new **ArrayList** (an implementation of the List interface), initially containing all the elements in c.

```
List<String> list = new ArrayList<String>(c);
```

The following shows the Collection interface.

```
public interface Collection<E> extends Iterable<E> {
  // Basic operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);          //optional
  boolean remove(Object element); //optional
  Iterator<E> iterator();

  // Bulk operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c); //optional
  boolean removeAll(Collection<?> c);        //optional
  boolean retainAll(Collection<?> c);        //optional
  void clear();                              //optional
```

```
    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The interface does about what you'd expect given that a Collection represents a group of objects. The interface has methods to tell you how many elements are in the collection (size, isEmpty), to check whether a given object is in the collection (contains), to add and remove an element from the collection (add, remove), and to provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

The Set Interface

A **Set** is a **Collection** that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The following is the Set interface.

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optional
    boolean remove(Object element); //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
```

```
    boolean removeAll(Collection<?> c);       //optional
    boolean retainAll(Collection<?> c);       //optional
    void clear();                             //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

The Java platform contains three general-purpose Set implementations: **HashSet**, **TreeSet**, and **LinkedHashSet**. **HashSet,** which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. **TreeSet,** which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than **HashSet**. **LinkedHashSet,** which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). **LinkedHashSet** spares its clients from the unspecified, generally chaotic ordering provided by **HashSet** at a cost that is only slightly higher.

Here's a simple but useful Set idiom. Suppose you have a **Collection c**, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a **Set** (which, by definition, cannot contain a duplicate), initially containing all the elements in c. It uses the standard conversion constructor in the **The Collection Interface .**

Here is a minor variant of this idiom that preserves the order of the original collection while removing duplicate element.

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a Set of the same generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c)
{
    return new LinkedHashSet<E>(c);
}
```

Set Interface Basic Operations

The size operation returns the number of elements in the **Set** (its *cardinality*). The **isEmpty()** method does exactly what you think it would. The add method adds the specified element to the Set if it's not already present and returns a boolean indicating whether the element was added. Similarly, the remove method removes the specified element from the Set if it's present and returns a boolean indicating whether the element was present. The iterator method returns an **Iterator** over the **Set**.

The following program takes the words in its argument list and prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated.

Bin>edit FindDups.java

```
import java.util.*;
public class FindDups
{
 public static void main(String[] args)
 {
     Set<String> s = new HashSet<String>();
     for (String a : args)
         if (!s.add(a))
            System.out.println("Duplicate detected: " + a);

    System.out.println(s.size() + " distinct words: " + s);
 }
}
```

Bin>javac FindDups.java
Bin>java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words: [i, left, saw, came]

Note that the code always refers to the **Collection** by its interface type (**Set**) rather than by its implementation type (**HashSet**). This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the Collection's implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the Set in the preceding example is **HashSet**, which makes no guarantees as to the order of the elements in the Set. If you want the program to print the word list in alphabetical order, merely change the Set's implementation type from **HashSet** to **TreeSet**. Making this trivial one-line change causes the command line in the previous example to generate the following output.

```
Bin>java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words: [came, i, left, saw]
```

We will discuss some of the Collection classes such as **ArrayList**, **LinkedList**, **TreeSet**, **Vector**, **Stack**.

The ArrayList class

java.util.ArrayList allows for expandable arrays, and is basically the same as the older the Collections Vector class. An **ArrayList** has these characteristics:

An **ArrayList** automatically expands as data is added.

Access to any element of an **ArrayList** is **O(1).** Insertions and deletions are **O(N)**.

An **ArrayList** has methods for inserting, deleting, and searching.

An **ArrayList** can be traversed using a ***foreach*** loop, iterators, or indexes.

Programmers are frequently faced with the choice of using a simple array or an **ArrayList**. If the data has a known number of elements or small fixed size upper bound, or where efficiency in using primitive types is important, arrays are often the best choice. However, many data storage problems are not that simple, and **ArrayList** (or one of the other Collections classes) might be the right choice.

Use **ArrayList** when there will be a large variation in the amount of data that you would put into an array. Arrays should be used only when there

is a constant amount of data. For example, storing information about the days of the week should use an array because the number of days in a week is constant. Use an array list for your email contact list because there is no upper bound on the number of contacts.

A possible disadvantage of **ArrayList** is that it holds only object types and not primitive types (eg, **int**). To use a primitive type in an **ArrayList**, put it inside an object or use of the *wrapper* classes (eg, **Integer**, **Double**, **Character**, ...). The wrapper classes are ***immutable***, so if you use, eg, **Integer**, you will not be able to change the integer value. In this case it may be more useful to define your own mutable class.

ArrayLists are implemented with an underlying array, and when that array is full and an additional element is added, a new, larger, array is allocated and the elements are copied from the old to the new. Because it takes time to create a bigger array and copy the elements from the old array to the new array, it is a slightly faster to create an **ArrayList** with a size that it will commonly be when full. Of course, if you knew the final size, you could simply use an array. However, for non-critical sections of code programmers typically don't specify an initial size.

Constructors	Meaning
ArrayList()	Constructs an empty list with an initial capacity of ten.
ArrayList(Collection<? extends E> c)	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity)	Constructs an empty list with the specified initial capacity.

Methods	Meaning
boolean **add**(E e)	Appends the specified element to the end of this list.
void **add**(int index, E element)	Inserts the specified element at the specified position in this list.
boolean **addAll**(Collection<? extends E> 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.

boolean **addAll**(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
void **clear**()	Removes all of the elements from this list.
Object **clone**()	Returns a shallow copy of this ArrayList instance.
boolean **contains**(Object o)	Returns true if this list contains the specified element.
void **ensureCapacity**(int minCapacity)	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.
E **get**(int index)	Returns the element at the specified position in this list.
int **indexOf**(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean **isEmpty**()	Returns true if this list contains no elements.
int **lastIndexOf**(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E **remove**(int index)	Removes the element at the specified position in this list.
boolean **remove**(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
protected void **removeRange**(int fromIndex, int toIndex)	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
E **set**(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int **size**()	Returns the number of elements in this list.
Object[**toArray**()]	Returns an array containing all of the
Object[**toArray**()]	Returns an array containing all of the elements in this list in proper sequence (from first to last element).

<T> T[]**toArray**(T[] a)	Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
void **trimToSize**()	Trims the capacity of this ArrayList instance to be the list's current size.

A program to demonstrate the ArrayList class

>edit arraylist1.java

```
import java.util.*;
class arraylist1
{
   public static void main(String argv[])
   {
         ArrayList l=new ArrayList();
         System.out.print("\n Size of l="+l.size());
         l.add("A");
         l.add("B");
         l.add("C");
         System.out.print("\n size of l="+l.size());
         System.out.print("\n values of l="+l);
         l.add(1,"z");
         l.remove("B");
         System.out.print("\n Size of l="+l.size());
         System.out.print ("\n values of l="+l);
   }
}
```

Bin>javac arraylist1.java

Bin>java arraylist1

```
Size of l=0
size of l=3
values of l=[A, B, C]
Size of l=3
values of l=[A, z, C]
```

Traversing Collections

There are two ways to traverse collections: (1) with **the for-each** construct and (2) by using **Iterators**.

for-each Construct

The **for-each** construct allows you to concisely traverse a collection or array using a for loop. The following code uses the **for-each** construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
        System.out.println(o);
```

Example:

>edit foreach1.java

```
import java.util.ArrayList;
class foreach1
{
   public static void main(String argv[])
   {
         ArrayList l=new ArrayList();
         l.add("A");
         l.add("B");
         l.add("C");
         System.out.print("\n objects in arraylist :");
         for (Object i : l)
           System.out.print("  " + i);
   }
}
```

Bin>javac foreach1.java
Bin>java foreach1

```
objects in arraylist :  A  B  C
```

Iterators

Iterators are used to traverse though a collection. Iterators allows to access individual elements of collection. Iterators are of two types: (1) **Iterator** (2) **ListIterator**

Iterator

An **Iterator** is an object that enables you to traverse through a collection and to remove elements from the collection selectively. **Iterator** allows to access elements of the collection sequentially in forward direction. You get an **Iterator** for a collection by calling its **iterator()** method. The following table show the methods of **Iterator** interface.

Methods	Meaning
Boolean **hasNext()**	Returns true if the iteration has more elements.
Object **next()**	Returns the next element in the iteration.
void **remove()**	Removes from the underlying collection the last element returned by the iterator (optional operation).

The **hasNext()** method returns true if the iteration has more elements, and the **next()** method returns the next element in the iteration. The **remove()** method removes the last element that was returned by next from the underlying Collection. The remove method may be called only once per call to next and throws an exception if this rule is violated.

A program to traverse an arraylist using Iterator.

Bin>edit iterator1.java

```
import java.util.*;
class iterator1
{
   public static void main(String argv[])
   {
         ArrayList l=new ArrayList();
         l.add("A");
         l.add("B");
         l.add("C");
         Iterator i=l.iterator();
         System.out.print("\n objects in arraylist :");
         while(i.hasNext())
         {
               System.out.print("  " + i.next() );
         }
   }
}
```

Bin>javac iterator1.java

Bin>java iterator1

```
objects in arraylist :  A  B  C
```

ListIterator

The **Iterator** allows to access elements of the list in sequential order. List also provides a richer iterator, called a **ListIterator**. An **ListIterator** for lists that allows the programmer to traverse the list in either direction(forward and backward), modify the list during iteration, and obtain the iterator's current position in the list. A **ListIterator** has no current element; its *cursor position* always lies between the element that would be returned by a call to **previous()** and the element that would be returned by a call to **next()**. In a list of length n, there are n+1 valid index values, from 0 to n, inclusive.

```
        Element(0)  Element(1)  Element(2)  ... Element(n)
            ^           ^           ^             ^
Index:      0           1           2            n+1
```

Note that the **remove()** and **set(Object)** methods are *not* defined in terms of the cursor position; they are defined to operate on the last element returned by a call to **next()** or **previous()**.

The following table contains the methods of **ListIterator** interface.

Methods	Meaning
void **add**(Object o)	Inserts the specified element into the list (optional operation).
boolean **hasNext**()	Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean **hasPrevious**()	Returns true if this list iterator has more elements when traversing the list in the reverse direction.
Object **next**()	Returns the next element in the list.
int **nextIndex**()	Returns the index of the element that would be returned by a subsequent call to next.
Object **previous**()	Returns the previous element in the list.
int **previousIndex**()	Returns the index of the element that would be returned by a subsequent call to previous.

void **remove**()	Removes from the list the last element that was returned by next or previous (optional operation).
void **set**(Object o)	Replaces the last element returned by next or previous with the specified element (optional operation).

The three methods that **ListIterator** inherits from **Iterator** (hasNext(), next(), and remove()) do exactly the same thing in both interfaces. The **hasPrevious()** and the **previous()** operations are exact analogues of **hasNext()** and **next()**. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

A program to traverse an arraylist using Iterator.

Bin>edit listiterator1.java

```
import java.util.*;
class listiterator1
{
   public static void main(String argv[])
   {
         ArrayList l=new ArrayList();
         l.add("A");
         l.add("B");
         l.add("C");
         System.out.print("\n objects in arraylist :");
         ListIterator li=l.listIterator();
         while(li.hasNext())
         {
               System.out.print("  " + li.next() );
         }
         System.out.print("\nObjects in arraylist in reverse:");
         while(li.hasPrevious())
         {
               System.out.print("  " + li.previous() );
         }
   }
}
```

```
Bin>javac listiterator1.java
Bin>java listiterator1
objects in arraylist : A B C
Objects in arraylist in reverse : C B A
```

The LinkedList class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List** interface. It provides a linked-list data structure. It has the two constructors, shown here:

Constructors	Meaning
LinkedList()	Constructs an empty list.
LinkedList(Collection<? extends E> c)	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection *c*.

In addition to the methods that it inherits, the **LinkedList** class defines some useful methods of its own for manipulating and accessing lists. To add elements to the start of the list, use **addFirst( )**; to add elements to the end, use **addLast( )**.

To obtain the first element, call **getFirst( )**. To retrieve the last element, call **getLast( )**. Their signatures are shown here:

To remove the first element, use **removeFirst( )**; to remove the last element, call **removeLast( )**. They are shown here:

The **LinkedList** class has methods that are shown here:

Methods	Meaning
boolean **add**(E e)	Appends the specified element to the end of this list.
void **add**(int index, E element)	Inserts the specified element at the specified position in this list.
boolean **addAll**(Collection<? extends E> 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.

boolean **addAll**(int index, Collection<? extends E> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
void **addFirst**(E e)	Inserts the specified element at the beginning of this list.
void **addLast**(E e)	Appends the specified element to the end of this list.
void **clear**()	Removes all of the elements from this list.
Object **clone**()	Returns a shallow copy of this LinkedList.
boolean **contains**(Object o)	Returns true if this list contains the specified element.
Iterator<E> **descendingIterator**()	Returns an iterator over the elements in this deque in reverse sequential order.
E **element**()	Retrieves, but does not remove, the head (first element) of this list.
E **get**(int index)	Returns the element at the specified position in this list.
E **getFirst**()	Returns the first element in this list.
E **getLast**()	Returns the last element in this list.
int **indexOf**(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int **lastIndexOf**(Object o)	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E> **listIterator**(int index)	Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
boolean **offer**(E e)	Adds the specified element as the tail

boolean **offerFirst**(E e)	Inserts the specified element at the front of this list.
boolean **offerLast**(E e)	Inserts the specified element at the end of this list.
E **peek**()	Retrieves, but does not remove, the head (first element) of this list.
E **peekFirst**()	Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
E **peekLast**()	Retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
E **poll**()	Retrieves and removes the head (first element) of this list
E **pollFirst**()	Retrieves and removes the first element of this list, or returns null if this list is empty.
E **pollLast**()	Retrieves and removes the last element of this list, or returns null if this list is empty.
E **pop**()	Pops an element from the stack represented by this list.
void **push**(E e)	Pushes an element onto the stack represented by this list.
E **remove**()	Retrieves and removes the head (first element) of this list.
E **remove**(int index)	Removes the element at the specified position in this list.
boolean **remove**(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
E **removeFirst**()	Removes and returns the first element from this list.
boolean **removeFirstOccurrence**(Object o)	Removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
E **removeLast**()	Removes and returns the last element from this list.

boolean **removeLastOccurrence**(Object o)	Removes the last occurrence of the specified element in this list (when traversing the list from head to tail).
E **set**(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int **size**()	Returns the number of elements in this list.
Object[]**toArray**()	Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[] **toArray**(T[] a)	Returns an array containing all of the elements in this list in proper sequence

The following program demonstrates several of the methods supported by LinkedList

Bin>edit linkedlist1.java

```
import java.util.*;
class linkedlist1
{
	public static void main(String args[])
	{
		LinkedList ll = new LinkedList();
		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);
		ll.remove("F");
		ll.remove(2);
		System.out.println("Contents of ll after deletion: " + ll);
		ll.removeFirst();
		ll.removeLast();
		System.out.println("ll after deleting first and last: " + ll);
		Object val = ll.get(2);
```

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

```
Bin>javac linkedlist1.java
Bin>java linkedlist1
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]
```

The Vector class

The **Vector** class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a **Vector** can grow or shrink as needed to accommodate adding and removing items after the **Vector** has been created.

Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement. An application can increase the capacity of a vector before inserting a large number of components; this reduces the amount of incremental reallocation. Vector class is synchronized.

The Iterators returned by Vector's iterator and **listIterator** methods are *fail-fast*: if the Vector is structurally modified at any time after the **Iterator** is created, in any way except through the Iterator's own remove or add methods, the **Iterator** will throw a **ConcurrentModificationException**. Thus, in the face of concurrent modification, the **Iterator** fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. The Enumerations returned by Vector's elements method are *not* fail-fast.

Constructors	Meaning
Vector()	Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
Vector(Collection<? extends E> c)	Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Vector(int initialCapacity)	Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
Vector(int initialCapacity, int capacityIncrement)	Constructs an empty vector with the specified initial capacity and capacity increment.

Methods	Meaning
boolean **add**(E o)	Appends the specified element to the end of this Vector.
void **add**(int index, E element)	Inserts the specified element at the specified position in this Vector.
boolean **addAll**(Collection<? extends E> c)	Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
boolean **addAll**(int index, Collection<? extends E> c)	Inserts all of the elements in the specified Collection into this Vector at the specified position.
void **addElement**(E obj)	Adds the specified component to the end of this vector, increasing its size by one.
int **capacity**()	Returns the current capacity of this vector.
void **clear**()	Removes all of the elements from this Vector.
Object **clone**()	Returns a clone of this vector.
boolean **contains**(Object elem)	Tests if the specified object is a component in this vector.
boolean **containsAll**(Collection<?> c)	Returns true if this Vector contains all of the elements in the specified Collection.
void **copyInto**(Object[] anArray)	Copies the components of this vector into the specified array.
E **elementAt**(int index)	Returns the component at the specified index.
Enumeration<E> **elements**()	Returns an enumeration of the components of this vector.

void **ensureCapacity**(int minCapacity)	Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
boolean **equals**(Object o)	Compares the specified Object with this Vector for equality.
E **firstElement**()	Returns the first component (the item at index 0) of this vector.
E **get**(int index)	Returns the element at the specified position in this Vector.
int **hashCode**()	Returns the hash code value for this Vector.
int **indexOf**(Object elem)	Searches for the first occurence of the given argument, testing for equality using the equals method.
int **indexOf**(Object elem, int index)	Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
void **insertElementAt**(E obj, int index)	Inserts the specified object as a component in this vector at the specified index.
boolean **isEmpty**()	Tests if this vector has no components.
E **lastElement**()	Returns the last component of the vector.
int **lastIndexOf**(Object elem)	Returns the index of the last occurrence of the specified object in this vector.
int **lastIndexOf**(Object elem, int index)	Searches backwards for the specified object, starting from the specified index, and returns an index to it.
E **remove**(int index)	Removes the element at the specified position in this Vector.
boolean **remove**(Object o)	Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
boolean **removeAll**(Collection<?> c)	Removes from this Vector all of its elements that are contained in the specified Collection.

void **removeAllElements**()	Removes all components from this vector and sets its size to zero.
boolean **removeElement**(Object obj)	Removes the first (lowest-indexed) occurrence of the argument from this vector.
void **removeElementAt**(int index)	Deletes the component at the specified index.
protected void **removeRange**(int fromIndex, int toIndex)	Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
boolean **retainAll**(Collection<?> c)	Retains only the elements in this Vector that are contained in the specified Collection.
E **set**(int index, E element)	Replaces the element at the specified position in this Vector with the specified element.
void **setElementAt**(E obj, int index)	Sets the component at the specified index of this vector to be the specified object.
void **setSize**(int newSize)	Sets the size of this vector.
int **size**()	Returns the number of components in this vector.
List<E> **subList**(int fromIndex, int toIndex)	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
Object[]**toArray**()	Returns an array containing all of the elements in this Vector in the correct order.
<T> T[]**toArray**(T[] a)	Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.
String **toString**()	Returns a string representation of this Vector, containing the String representation of each element.
void **trimToSize**()	Trims the capacity of this vector to be the vector's current size.

A program to demonstrate methods of Vector class

Bin>edit vector1.java

```
import java.util.*;
class vector1
{
    public static void main(String[] args)
    {
        Vector  v = new Vector();
         v.add(100);
        v.add(new Integer(200));
        v.add("Apex");
        v.add(2, new Integer(30));
        System.out.println("the elements of v: " + v);
        System.out.println("The size of v are: " + v.size());
        System.out.println("The elements at position 2
                            is: " + v.elementAt(2));
        System.out.println("The first element of v
                            is: " + v.firstElement());
        System.out.println("The last element of v is: "+
                            v.lastElement());
        v.removeElementAt(2);
        Enumeration e=v.elements();
        System.out.println("The elements of v: " + v);
        while(e.hasMoreElements())
        {
              System.out.println("The elements are:" +
                            e.nextElement());
        }
    }
}
```

Bin>javac vector1.java

Bin>java vector1

```
the elements of v: [100, 200, 30, Apex]
The size of v are: 4
The elements at position 2 is: 30
The first element of v is: 100
The last element of v is: Apex
The elements of v: [100, 200, Apex]
The elements are: 100
The elements are: 200
The elements are: Apex
```

The Stack class

The **Stack** class represents a last-in-first-out (**LIFO**) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

When a stack is first created, it contains no items.

Constructors	Meaning
Stack()	Creates an empty Stack.

Methods	Meaning
boolean **empty**()	Tests if this stack is empty.
Object **peek**()	Looks at the object at the top of this stack without removing it from the stack.
Object **pop**()	Removes the object at the top of this stack and returns that object as the value of this function.
Object **push**(Object item)	Pushes an item onto the top of this stack.
int	Returns the 1-based position where an object is on this stack.

A program to demonstrate the methods of Stack class

Bin>edit stack1.java

```
import java.util.*;
class stack1
{
        public static void main(String[] args)
        {
        Stack stack=new Stack();
        stack.push(new Integer(10));
        stack.push("a");
        System.out.println("The contents of Stack is" + stack);
        System.out.println("The size of an Stack is" + stack.size());
        System.out.println("The number poped out is" + stack.pop());
        System.out.println("The number poped out is " + stack.pop());
```

```
        //System.out.println("The number poped out is" + stack.pop());
        System.out.println("The contents of stack is" + stack);
        System.out.println("The size of an stack is" + stack.size());
        }
}
```

Bin>javac stack1.java
Bin>java stack1

```
The contents of Stack is[10, a]
The size of an Stack is2
The number poped out isa
The number poped out is 10
The contents of stack is[]
The size of an stack is0
```

The TreeSet Class

This class implements the **Set** interface, backed by a **TreeMap** instance. It creates a collection that uses a tree for storage. It stores the objects in sorted, ascending order. Accessing and retrieval of elements is faster as compared to other collection classes. The **TreeSet** is best to use, when you want to store large amount of data in sorted order.

Constructors	Meaning
TreeSet()	Constructs a new, empty set, sorted according to the elements' natural order.
TreeSet(Collection c)	Constructs a new set containing the elements in the specified collection, sorted according to the elements' *natural order.*
TreeSet(Comparator c)	Constructs a new, empty set, sorted according to the specified comparator.
TreeSet(SortedSet s)	Constructs a new set containing the same elements as the specified sorted set, sorted according to the same ordering.

Methods	Meaning
boolean **add**(Object o)	Adds the specified element to this set if it is not already present.
boolean **addAll**(Collection c)	Adds all of the elements in the specified collection to this set.

void **clear**()	Removes all of the elements from this set.
Object **clone**()	Returns a shallow copy of this TreeSet instance.
Comparator **comparator**()	Returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering.
boolean **contains**(Object o)	Returns true if this set contains the specified element.
Object **first**()	Returns the first (lowest) element currently in this sorted set.
SortedSet **headSet**(Object toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
boolean **isEmpty**()	Returns true if this set contains no elements.
Iterator **iterator**()	Returns an iterator over the elements in this set.
Object **last**()	Returns the last (highest) element currently in this sorted set.
boolean **remove**(Object o)	Removes the specified element from this set if it is present.
int **size**()	Returns the number of elements in this set (its cardinality).
SortedSet **subSet**(Object fromElement, Object toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet **tailSet**(Object fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

A program to demonstrate TreeSet class

>edit treeset1.java

```
import java.util.Iterator;
import java.util.TreeSet;

public class treeset1
{

   public static void main(String argv[])
   {
     TreeSet ts=new TreeSet();

     ts.add("b");
     ts.add("a");
     ts.add("d");
     ts.add("c");
     System.out.print("\n ts =" +ts);

     Iterator it=ts.iterator();

     System.out.print("\n Elements of ts :\n" );
     while(it.hasNext())
     {
       String value=(String)it.next();
       System.out.println("Value :"+value);
     }
   }
}
```

Bin>javac treeset1.java

Bin>java treeset1

```
ts =[a, b, c, d]
Elements of ts :
Value :a
Value :b
Value :c
Value :d
```