Module: Core Java
Session 18: Basics of Collection Framework

- Introduction to the Collections Framework
- Collection Interfaces
- Iterator Interface
- Group Operations
- AbstractCollection Class
- Collection Framework Design Concerns
- Set Interface
- HashSet, TreeSet Classes
- List Interface
- Map interface
- Sorting

Objective

At the end of this chapter, you will be able to:

- Learn about the implementation of Collection Interfaces:
- Understand the Iterator Interface and Group Operation
- Know in details about the AbstractCollection Class
- Prepare yourself for Collection Framework Design Concerns
- Gain knowledge about the Set Interface
- Get some ideas about the HashSet, TreeSet Classes
- Deal with List Interface, Map interface and Sorting
- Have a basic idea about Historical Collection Classes such as Dictionary, Hashtable and Properties Classes

Introduction

Data encapsulation is one of the important features of OOPs. But, this doesn't mean the data organization inside classes is not important. The way you structure your data depends on your requirement and the problem you need to solve. Does your class need to search for items quickly? Does it require sorted elements and rapid inserts and deletions of elements? Does it need an array-like structure with random-access ability that can grow at run time? The way data is structured inside classes is very important to improve the performance.

This chapter tells you how Java helps you structure your data needed for programming.

Prior to JDK1.2, only a few classes were provided for data structures by the standard library. Vector, Stack, HashTable, BitSet and Enumeration interface are some of them. Enumeration interface provided an abstract mechanism to access elements in an arbitrary container. It takes time and skill to come up with a comprehensive collection class library.

With the advent of JDK 1.2 the Standard Library supplied a full-fledged set of data structures. The designers had faced a number of conflicting design decisions. Hence they wanted the library to be small and easy to learn. They wanted the benefit of "generic algorithms" that STL pioneered and not the complexity of the "Standard Template Library" (or STL) of C++,. They wanted the new framework to accommodate the legacy classes. They had to make some hard choices, and came up with a number of distinctive design decisions. This chapter takes you through the basic design of the Java collections framework, let you know how to make it work, and reasons behind some of its controversial features.

Just like other data structure libraries, the Java collection library also separates *interfaces* and *implementations.*

Collections Overview

This chapter deals with one of the powerful subsystems: Collections. Collections provide a well-defined set of interfaces and classes to store and manipulate groups of data as a single unit. This unit is called a collection.

The framework provides a convenient API to many of the abstract datatypes like maps, sets, lists, trees, arrays, hashtables and other collections. The Java classes in the Collections Framework encapsulate both the data structures and the algorithms associated with these abstractions because of their object-oriented design.

The framework provides a standard programming interface to many of the most common abstractions, without complicating it with too many procedures and interfaces. The programmer can easily define higher-level data abstractions like stacks and queues with the help of a variety of operations supported by the collections framework.

The major goals of the Collections Framework are:

- High-performance.
- Implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
- Should avoid coding of these "data engines" manually.

- Should allow different types of collections to work in a similar manner.
- Should have a high degree of interoperability.
- Should be easy to extend and/or adapt a collection.

Towards this end, the entire collections framework is designed around a set of standard interfaces.

Several standard implementations (such as **LinkedList**, **HashSet**, and **TreeSet**) of these interfaces are ready to use. You can also use your own collections. Various special-purpose implementations are created for your convenience, and some partial implementations are provided to make it easier to have user-defined Collection class. Mechanisms were added to integrate standard arrays into the collections framework.

Algorithms are another major part of the collection mechanism, which operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections. Each collection class need not implement its own version. A standard means of manipulating collections is provides by these algorithms.

Iterator interface is another item created by the collections framework. It provides a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of *enumerating the contents of a collection*. Because each collection implements **Iterator**, the elements of any collection class can be accessed through the methods defined by **Iterator**. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

In addition to collections, several map interfaces and classes are defined within the framework. *Maps* store key/value pairs. Though maps are not actually collections, they are completed integrated with collections. In the language of the collections framework, you can obtain a *collection-view* of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection.

The collection mechanism was retrofitted to some of the original classes defined by **java.util** so that they too could be integrated into the new system. Although the addition of collections altered the architecture of many of the original utility classes, it did not cause the deprecation of any. Collections simply provide a better way to do many things.

One last thing: familiarity to C++ will help you to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls it ,a collection.

Collection Interfaces

The Collections Framework is made up of a set of interfaces for working with groups of objects. The different interfaces describe the different types of groups. Once you are clear with the interfaces, you can easily understand the framework. While you always need to create specific implementations of the interfaces, access to the actual collection should be restricted to the use of the interface methods, thus allowing you to change the underlying data structure, without altering the rest of your code. The following diagrams show the framework interface hierarchy.

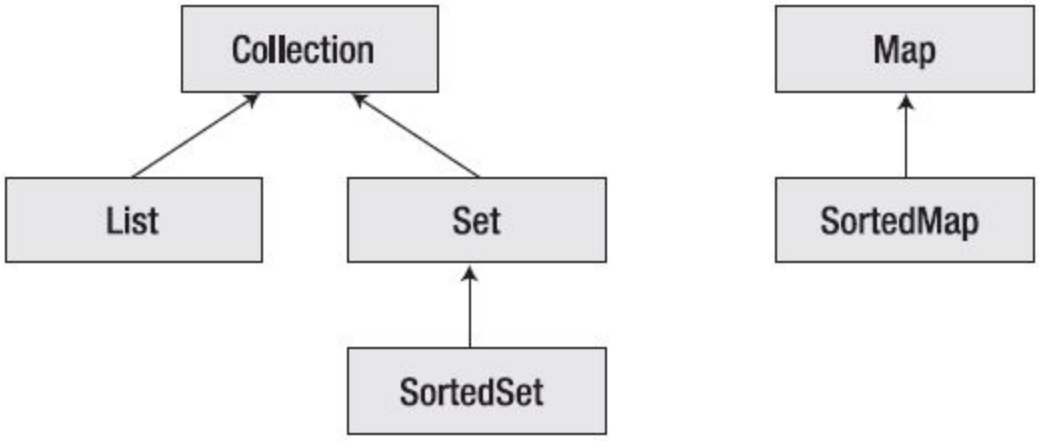

Fig 1:The hierarchy of collection interfaces.

The root of the hierarchy of the collections interfaces is the **Collection** interface. It is also referred to as the superinterface of the collections. There is another kind of collection called maps, which are represented by the superinterface **Map**, which is not derived from the Collection interface. Both kinds of collections interfaces are shown in Fig 1 above.

The List and Set interfaces extend from the Collection interface, and there is no direct implementation of the Collection interface. Some of characteristics of the subinterfaces of Collection (that is, List and Set) and of Map are as follows:

- **List**: An ordered collection of data items; i.e. The position of each data item is defined and known. A list can have duplicate elements.

  ArrayList, LinkedList, and Vector are the classes that implement the List interface.

- **Map**: An object that maps keys to values: each key can map to at most one value. Maps cannot have duplicate keys.

  HashMap and HashTable are some of the classes that implement the Map interface. Duplicate keys are not allowed in maps but duplicate values are allowed.

- **Set**: A collection of unique items; i.e. there are no duplicates.

  HashSet and LinkedHashSet are examples of classes that implement the Set interface.

Implementations of Collections Interfaces:

Classes, which implement the interfaces and represent reusable data structures to hold the data items, are the Implementations of the collections interfaces. As you know, the purpose of a collection is to represent a group of related objects, known as its elements. Depending upon the application requirements, these related data items could be grouped together in various ways.

Following Fig 2 shows the hierarchy of collection and map implementations.

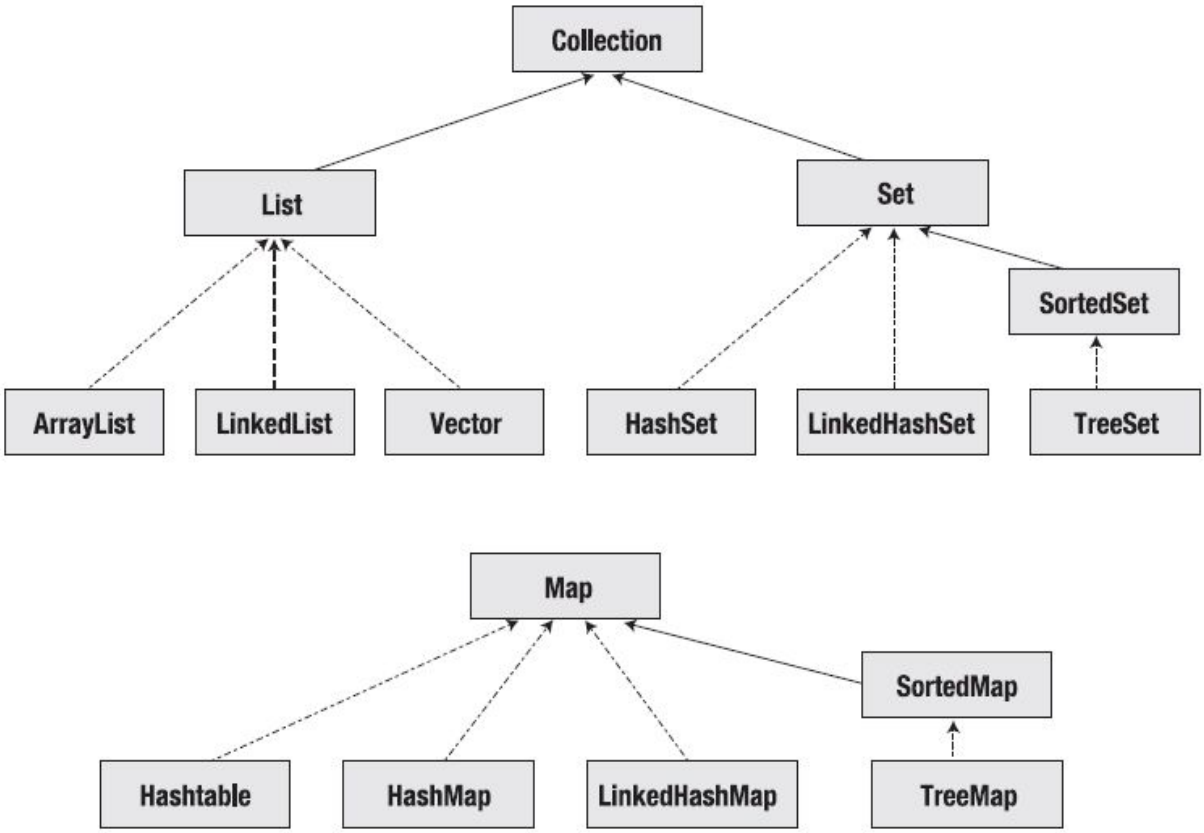

Fig 2:The hierarchy of collection and map implementations

As per the application requirements, duplicate elements are allowed with some collections. Some are ordered and others are unordered. There are certain restrictions on elements that an implementation can contain, such as no null elements, and there can be restrictions on the types of their elements. Consider the following features. The implementation you choose depends on which of them is important for your application.

• **Performance**: It refers to the time taken by a specific operation on the data as a function of the number of data items in the structure, such as accessing a given data item (search) and inserting or deleting an item in the data structure. In some data structures, the search and insertion/deletion may have opposing performance requirements. That means if a data structure offers fast search, the insertion/deletion may be slow.

• **Ordered/sorted**: A data structure is said to be ordered if the data items are in a specific order. For example, an array is an ordered data structure because the data items are ordered by index; that is, we can refer to the data items with their index, such as the seventh element. A data structure is said to be sorted if the data items are ordered either by ascending order or descending order of their values. So, by definition, a sorted data structure is an ordered data structure, but not vice versa.

• **Uniqueness of items:** If you require each data item of the data structure to be uniquely identifiable, or on the contrary, you want to allow duplicate items.

• **Synchronized:** Implementations are synchronized when they provide thread safety (i.e. they can be run in a multithreaded environment). However, need not be considered while choosing an implementation, because you can always use synchronization methods from the Collections class even if the implementation is unsynchronized.

The characteristics of different implementations of the collections interfaces are summarized in the following Table1:

Class	Interface	Duplicates Allowed?	Ordered/Sorted	Synchronized
ArrayList	List	Yes	Ordered by index Not sorted	No
LinkedList	List	Yes	Ordered by index Not sorted	No
Vector	List	Yes	Ordered by index Not sorted	Yes
HashSet	Set	No	Not ordered Not sorted	No
LinkedHashSet	Set	No	Ordered by insertion Not sorted	No
TreeSet	Set	No	Sorted either by natural order or by your comparison rules	No
HashMap	Map	No	Not ordered Not sorted	No
LinkedHashMap	Map	No	Ordered	No
Hashtable	Map	No	Not ordered Not sorted	Yes
TreeMap	Map	No	Sorted either by natural order or by your comparison rules	No

Table 1: Implementations of the Collections Interfaces characteristics

Usage of Pre-defined Methods of Collection Interface:

The **Collection** interface is used to represent any group of objects, or elements. The interface is used when you want to work with a group of elements in a very generalized manner.

Below is a list of the public methods of **Collection**:

- boolean add(Object element)

- boolean add(Collection col)
- void clear()
- boolean contains(Object element)
- boolean containsAll(Collection col)
- boolean equals(Object element)
- int hashcode()
- Iterator iterator()
- boolean remove(Object element)
- boolean removeAll(Collection col)
- boolean retainAll(Collection col)
- int size()
- Object[] toArray()
- Object[] toArray(Object[] array)

The interface supports basic operations like adding and removing. When an element is removed, only an instance of the element in the collection is removed, if present.

- boolean add(Object element)
- boolean remove(Object element)

The Collection interface also supports query operations:

- int size()
- boolean isEmpty()
- boolean contains(Object element)
- Iterator iterator()

Iterator Interface

The **iterator()** method of the **Collection** interface returns an **Iterator** interface type. An **Iterator** is similar to the **Enumeration** interface, which will be discussed later. You can traverse a collection from start to finish and safely remove elements from the underlying **Collection** with the help of the iterator() method:

boolean hasNext()

Object next()

void remove()

The remove() method is optionally supported by the underlying collection. Its function is to remove the element returned by the last next() call. To illustrate, the following shows the use of the Iterator interface for a general Collection:

```
//DEMONSTRATE ITERATOR TO ACCESS A COLLECTION
import java.util.*;

class iteratordemo
{
public static void main(String args[])
{
ArrayList al=new ArrayList();
        al.add("pehla");
        al.add("dusra");
        al.add("teesra");
        al.add("chautha");
        al.add("panchwa");
        al.add("chhatha");
        al.add("sathwa");
        al.add("athwa");
        al.add("nawa");
        al.add("daswa");
//Use iterator to display contents of al.
System.out.println("Original contents of al :");
        Iterator itr = al.iterator();
while(itr.hasNext())
{
Object element=itr.next();
System.out.print(element+" ");
}
System.out.println();
//Modify objects being created.
        ListIterator litr = al.listIterator();
while(litr.hasNext())
{
Object element=litr.next();
litr.set(element+"+ ");
}
System.out.println("Modified contents of al :");
itr = al.iterator();
while(itr.hasNext())
```

```
{
Object element=itr.next();
System.out.print(element+" ");
}
System.out.println();
//Now display the list backwards.
System.out.println("Modified list backwards :");
while(litr.hasPrevious())
{
Object element=litr.previous();
System.out.print(element+" ");
}
System.out.println();
}
}
```

Group Operations

Group operations are tasks done on groups of elements or the entire collection at once:

- boolean containsAll(Collection collection)
- boolean addAll(Collection collection)
- void clear()
- void removeAll(Collection collection)
- void retainAll(Collection collection)

The containsAll() method tells you whether the current collection contains all the elements of another collection, a subset. The remaining methods are optional, in that a specific collection might not support the altering of the collection.

The addAll() method ensures the addition of all elements from another collection to the current collection, usually a union.

The clear() method removes all elements from the current collection.

The removeAll() method is like clear() but only removes a subset of elements.

The retainAll() method is similar to the removeAll() method, but does what might be perceived as the opposite. It removes from the current collection those elements not in the other collection, an intersection.

The remaining two interface methods, which convert a Collection to an array, will be discussed later.

AbstractCollection Class

The AbstractCollection class provides the foundation for the concrete collection framework classes. While you can implement all the methods of the Collection interface yourself, the AbstractCollection class provides implementations for all the methods, except for the iterator() and size() methods, which are implemented in the appropriate subclass. Optional methods like add() will throw an exception if the subclass doesn't override the behavior.

Collection Framework Design Concerns

While creating the Collections Framework, the Sun development team had to provide flexible interfaces that manipulated groups of elements. To keep the design simple, the interfaces define all the methods an implementation class may provide. However, some of the interface methods are optional. Because an interface implementation must provide implementations for all the interface methods, there needed to be a way for a caller to know if an optional method is not supported. The manner the framework development team chose to signal callers when an optional method is called was to throw an UnsupportedOperationException exception. If while using a collection, an UnsupportedOperationException is thrown while trying to add to a read-only collection, then it indicates that the operation failed because it is not supported.Yhe UnsupportedOperationException class is an extension of the RuntimeException class.

In addition to handling optional operations with a runtime exception, the iterators for the concrete collection implementations are fail-fast. That means that if you are using an Iterator to traverse a collection while underlying collection is being modified by another thread, then the Iterator fails immediately by throwing a ConcurrentModificationException (another RuntimeException). That means the next time an Iterator method is called, and the underlying collection has been modified, the ConcurrentModificationException exception gets thrown.

Set Interface

The Set interface defines a set and extends the Collection interface. It forbids duplicates within the collection. All the original methods are present and no new methods are introduced. The concrete Set implementation classes rely on the equals() method of the object added to check for equality.

- boolean add(Object element)
- boolean addAll(Collection col)
- void clear()
- boolean contains(Object element)
- boolean containsAll(Collection col)
- boolean equals(Object object)
- int hashCode()
- Iterator iterator()
- boolean remove(Object element)
- boolean removeAll(Collection col)

- boolean retainAll(Collection col)
- int size()
- Object[] toArray()

HashSet, TreeSet Classes

HashSet and TreeSet are the two general-purpose implementations of the Set interface provided by the Collections framework. Generally, you will use a HashSet for storing your duplicate-free collection. In order to improve the efficiency, objects added to a HashSet need to implement the hashCode() method in a manner that properly distributes the hash codes. Most of the system classes override the default hashCode() implementation in Object. You have to override hashCode() when creating your own classes to add to a HashSet. You will find the TreeSet implementation to be useful when you need to extract elements from a collection in a sorted manner. In order to work properly, elements added to a TreeSet must be sortable. The Collections Framework adds support for Comparable elements and will be covered in detail later. As of now, assume a tree knows how to keep elements of the java.lang wrapper classes sorted. It is generally faster to add elements to a HashSet and then convert the collection to a TreeSet for sorted traversal.

Tuning the initial capacity and the load factor you will be able to make optimal usage of HashSet space. The TreeSet has no tuning options, as the tree is always balanced, ensuring log(n) performance for insertions, deletions, and queries.

Both HashSet and TreeSet implement the Cloneable interface.

HashSet Usage Example

```
import java.util.*;
class hashsetdemo
{
public static void main(String args[])
{
HashSet hs=new HashSet();
        hs.add("TERA");
        hs.add("JADOO");
        hs.add("CHAL");
        hs.add("GAYA");

System.out.println(hs);
}
}
```

TreeSet Usage example:

```
import java.util.*;
class treesetdemo
{
public static void main(String args[])
{
TreeSet t=new TreeSet();
        t.add("t");
        t.add("j");
        t.add("c");
        t.add("g");


System.out.println(t);
}
}
```

AbstractSet Class

To ensure two equal sets return the same hash code, the AbstractSet class overrides the equals() and hashCode() methods. If both the sets are the same size and contain the same elements, then they are said to be equal. By definition, the hash code for a set is the sum of the hash codes for the elements of the set. Thus, irrespective of what the internal ordering of the sets is, same hash code is reported by the two equal sets.

List Interface

The **List** interface extends the **Collection** interface to define an ordered collection and it permits duplicates. The interface adds position-oriented operations, and also the ability to work with just a part of the list. Below is a list of methods provided by the List Interface:

- boolean add(Object element)
- void add(int index, Object element)
- boolean addAll(Collection collection)
- boolean addAll(int index, Collection collection)
- void clear()
- boolean contains(Object element)
- boolean containsAll(Collection collection)
- boolean equals(Object object)
- Object get(int index)

- int hashCode()
- Iterator iterator()
- int lastIndexOf(Object element)
- ListIterator listIterator()
- ListIterator listiterator(int startIndex)
- boolean remove(Object element)
- Object remove(int index)
- boolean removeAll (Collection collection)
- boolean retainAll(Collection collection)
- Object set(int index, Object element)
- int size()
- List sublist(int fromIndex, int toIndex)
- Object[] toArray()
- Object[] toArray(Object[] array)

The position-oriented operations include the ability to insert an element or **Collection**, get an element, as well as remove or change an element. Searching for an element in a **List** can be started from the beginning or end and will report the position of the element, if found. Some of the methods to implement position-oriented operations are:

- void add(int index, Object element)
- boolean addAll(int index, Collection collection)
- Object get(int index)
- int indexOf(Object element)
- int lastIndexOf(Object element)
- Object remove(int index)
- Object set(int index, Object element)

The **List** interface also helps working with a subset of the collection, and iterating through the entire list in a position friendly manner:

- ListIterator listIterator()
- ListIterator listIterator(int startIndex)
- List subList(int fromIndex, int toIndex)

It is important to mention that the element at **fromIndex** is in the sub list in working with **subList(**), but the element at **toIndex** is not. This loosely maps to the following **for**-**loop** test cases:

```
for (int i=fromIndex; i<toIndex; i++) {
```

```
  // process element at position i
}
```

In addition, it should be mentioned that changes to the sublist like **add()**, **remove()**, and **set()** calls have an effect on the underlying **List**.

- **ListIterator Interface**

To support bi-directional access, as well as adding or changing elements in the underlying collection, Iterator interface is extended to ListIterator interface. listIterator() method returns an object of **listIterator** class type.

- void add(Object element)
- boolean hasNext()
- boolean hasPrevious()
- Object next()
- int nextIndex()
- Obejct previous()
- int previousIndex()
- void remove()
- void set(Obejct element)

The code below illustrates the looping backwards through a list. Notice that the ListIterator is originally positioned beyond the end of the list [list.size()], as the index of the first element is 0.

```
List list = ...;
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
  Object element = iterator.previous();
  // Process element
}
```

Generally, ListIterator is not used to alternate between going forward and backward in one iteration through the elements of a collection. While technically possible, one needs to know that calling next() immediately after previous() results in the same element being returned. The same thing happens when you reverse the order of the calls to next() and previous().

In case of add() operation, adding an element results in the new element being added immediately prior to the implicit cursor. Thus, calling previous() after adding an element would return the new element and calling next() would have no effect, returning what would have been the next element prior to the add operation.

- **ArrayList, LinkedList Classes**

There are two general-purpose List implementations in the Collections Framework: ArrayList and LinkedList. Based on your specific needs, you can choose between the two list implementations. Suppose, random access is what you require, without inserting or removing elements from any place other than the end, then ArrayList offers the optimal collection.

However, if your requirement is frequent addition and removal of elements from the middle of the list and only access the list elements sequentially, then, LinkedList offers the better implementation.

Both the general-purpose implementations, ArrayList and LinkedList, implement the Cloneable interface. Additionally, to work with the elements at the ends of the list, several methods are provided by LinkedList. (Only the new methods are shown in the following list):

- void addFirst(Object element)
- void addLast(Object element)
- Object getFirst()
- Object getLast()
- Obejct removeFirst()
- Obejct removeLast()

Below is a program to demonstrate ArrayList usage:

```
import java.util.* ;
class ArrayListDemo
{
public static void main(String args[])
{
ArrayList a1 = new ArrayList();
System.out.println("Initital Size Ofa1 " +a1.size()) ;
//add elements
a1.add("sql");
a1.add("niit");
a1.add("aptech");
a1.add("ssi");
a1.add("cmc");
a1.add("pannet");
a1.add("tulec");
a1.add("ignou");
a1.add("iddco");

System.out.println(" Size Of a1  After addition " +a1.size()) ;
// display array list
System.out.println("content of  a1 " +a1) ;
```

```
a1.remove("tulec");
a1.remove(3);
System.out.println(" Size Of a1  After deletion " +a1.size()) ;
System.out.println(" content of a1 " +a1) ;
}
}
```

Below is the program to implement Linkedlist:

```
import java.util.*;
class linkedlistdemo
{
public static void main(String args[])
{
LinkedList ll=new LinkedList();
System.out.println("Initial size of ll :"+ll.size());
//add elements
        ll.add("sql");
        ll.add("niit");
        ll.add("nimt");
        ll.add("aptech");
        ll.add("first");
        ll.add("ignou");
        ll.add("idco");
        ll.add("narimann");
        ll.add("telco");
        ll.add("officenet");
        ll.add("hindalco");
        ll.add("nalco");
        ll.addLast("last mein rahe gaya");
        ll.addFirst("pehle main aa gaya");
ll.add(2,"teen number pe ghus gaya");
System.out.println("Size of ll after addition :"+ll.size());

//display array list
System.out.println("Contents of ll :"+ll);
ll.removeFirst();
```

```
ll.removeLast();
System.out.println("Size of ll after this :"+ll.size());
System.out.println("Contents of ll :"+ll);

//get & set the vals
Object val=ll.get(2);
ll.set(2,(String)val+" Changed");
System.out.println("ll after changed :"+ll);
}
}
```

You can easily treat the LinkedList as a stack, queue, or other end-oriented data structure by using new methods.

```
LinkedList queue = ...;
queue.addFirst(element);
Object object = queue.removeLast();
LinkedList stack = ...;
stack.addFirst(element);
Object object = stack.removeFirst();
```

The Vector and Stack classes will be discussed in later sections.

- **List Usage Example**

To illustrate the use of the concrete List classes, see the following program. The first part creates a List backed by an ArrayList. After filling the list, specific entries are retrieved. The LinkedList part of the example treats the LinkedList as a queue, adding things at the beginning of the queue and removing them from the end.

```
import java.util.*;

public class Listdemo {
  public static void main(String args[]) {
    List list = new ArrayList();
    list.add("Sunday");
    list.add("Monday");
    list.add("Tuesday");
    list.add("Monday");
```

```
        list.add("Wednesday");
        System.out.println(list);
        System.out.println("2: " + list.get(2));
        System.out.println("0: " + list.get(0));
        LinkedList queue = new LinkedList();
        queue.addFirst("Sunday");
        queue.addFirst("Monday");
        queue.addFirst("Tuesday");
        queue.addFirst("Monday");
        queue.addFirst("Wednesday");
        System.out.println(queue);
        queue.removeLast();
        queue.removeLast();
        System.out.println(queue);
    }
}
```

Running the program produces the following output. Notice that, unlike Set, List permits duplicates.

```
[Sunday, Monday, Tuesday, Monday, Wednesday]
2: Tuesday
0: Sunday
[Wednesday, Monday, Tuesday, Monday, Sunday]
[Wednesday, Monday, Tuesday]
```

- **AbstractList and AbstractSequentialList Classes**

AbstractList and AbstractSequentialList are two abstract List implementations classes. Just Like the AbstractSet class, they override the equals() and hashCode() methods to ensure two equal collections return the same hash code. If they are the same size and contain the same elements in the same order, then the two sets are said to be equal. The hashCode() implementation is specified in the List interface definition and implemented here.

Apart from the equals() and hashCode() implementations, AbstractList and AbstractSequentialList provide partial implementations of the remaining List methods. For random-access and sequential-access data sources, respectively, they help creating concrete list implementations easily. Based on which behavior you want to support, you can choose between the set of methods.

The table below shows methods that need to be implemented. But, you never to provide the implementation of the Iterator iterator() method.

	AbstractList	**AbstractSequentialList**
unmodifiable	Object get(int index) int size()	ListIterator listIterator(int index) - boolean hasNext() - Object next() - int nextIndex() - boolean hasPrevious() - Object previous() - int previousIndex() int size()
modifiable	unmodifiable + Object set(int index, Object element)	unmodifiable + ListIterator - set(Object element)
variable-size and modifiable	modifiable + add(int index, Object element) Object remove(int index)	modifiable + ListIterator - add(Object element) - remove()

Two constructors, a no-argument one and one that accepts another Collection, should also be provided.

Map Interface

The Map interface starts off its own interface hierarchy, for maintaining key-value associations. It is not an extension to Collection interface. According to it's definition, the interface describes a mapping from keys to values, without duplicate keys.

- void clear()
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- Set entrySet()
- Object get(Object key)
- boolean isEmpty()
- Set keySet()
- Object put(Object key, Object value)
- void putAll(Map mapping)
- Object remove(Object key)

- int size()
- Collection values()

The interface methods can be broken down into three sets of operations: altering, querying, and providing alternative views.

The alteration operations allow you to add and remove key-value pairs from the map even when the key and value can be null. However, you should not add a Map to itself as a key or value.

- Object put(Object key, Object value)
- Object remove(Object key)
- void putAll(Map mapping)
- void clear()

The query operations allow you to check on the contents of the map:

- Object get(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()

The last set of methods allows you to work with the group of keys or values as a collection.

- public Set keySet()
- public Collection values()
- public Set entrySet()

Since the collection of keys in a map must be unique, you get a Set back. Since the collection of values in a map may not be unique, you get a Collection back. The last method returns a Set of elements that implement the Map.Entry interface, described next.

- **Map.Entry Interface**

The entrySet() method of Map returns a collection of objects that implement Map.Entry interface. Each object in the collection is a specific key-value pair in the underlying Map.

- boolean equals(Object object)
- Object getKey()
- Object getValue()
- int hashCode()
- Object setValue(Object value)

Iterating through this collection, you can get the key or value, as well as change the value of each entry. However, the set of entries becomes invalid, causing the iterator behavior to be

undefined, if the underlying Map is modified outside the setValue() method of the Map.Entry interface.

- **HashMap, TreeMap Classes**

HashMap and TreeMap are the two general-purpose Map implementations provided by the Collections Framework. As with all the concrete implementations, which implementation you use depends on your specific needs. The HashMap offers the best alternative for inserting, deleting, and locating elements in a Map. However, if you need to traverse the keys in a sorted order, then TreeMap is a better alternative. Depending upon the size of your collection, it may be faster to add elements to a HashMap, and then convert the map to a TreeMap for sorted key traversal. Using a HashMap requires that the class of key added have a well-defined hashCode() implementation. With the TreeMap implementation, elements added to the map must be sortable.

Tuning the initial capacity and load factor will help you make optimal use of HashMap space. The TreeMap has no tuning options, as the tree is always balanced.

The Cloneable interface can be implemented with both HashMap and TreeMap.

The Hashtable and Properties classes are historical implementations of the Map interface.

- **Map Usage Example**

The following program demonstrates the use of the concrete Map classes.

First let us see the usage of HashMap class:

HashMap Example:

```
import java.util.*;
class HashMapDemo
{
public static void main(String args[])
{
HashMap hm = new HashMap();
hm.put("amarendra mohanty",new Double(5000));
hm.put("sarada satpathy",new Double(4500));
hm.put("pravat pala",new Double(6000));
hm.put("mitrabhanu tripathy",new Double(7000));

Set set =hm.entrySet();
Iterator i =set.iterator();
while (i.hasNext())
{
Map.Entry me = (Map.Entry)i.next();
System.out.println(me.getKey()+":");
```

```
System.out.println(me.getValue());
}
System.out.println();
double balance = ((Double) hm.get("amarendra mohanty")).doubleValue();
hm.put("amarendra mohanty", new Double ( balance +1000));
System.out.println("amarendra mohanty new balance is " +hm.get("amarendra
mohanty"));
}
}
```

Now, let us see the usage of a TreeMap class implementation:

TreeMap Example:

```
import java.util.*;
class TreeMapDemo
{
public static void main(String args[])
{
TreeMap hm = new TreeMap();
hm.put("mitrabhanu tripathy ",new Double(5000));
hm.put("sarada satpathy",new Double(4500));
hm.put("amarendra mohanty",new Double(6000));
hm.put("pravat pala",new Double(7000));

Set set =hm.entrySet();
Iterator i =set.iterator();
while (i.hasNext())
{
Map.Entry me = (Map.Entry)i.next();
System.out.println(me.getKey()+":");
System.out.println(me.getValue());
}
System.out.println();
double balance = ((Double) hm.get("pravat pala")).doubleValue();
hm.put("pravat pala", new Double ( balance +5000));
System.out.println("pravat pala new balance is " +hm.get("amarendra mohanty"));
}
}
```

- **AbstractMap Class**

Similar to the other abstract collection implementations, the AbstractMap class overrides the equals() and hashCode() methods to ensure two equal maps return the same hash code. Two maps are equal if they are the same size, contain the same keys, and each key maps to the same value in both maps. By definition, the hash code for a map is the sum of the hash codes for the elements of the map, where each element is an implementation of the Map.Entry interface. Thus, no matter what the internal ordering of the maps, two equal maps will report the same hash code.

- **WeakHashMap Class**

A special-purpose implementation of Map called A WeakHashMap is provided for storing only weak references to the keys. This allows for the key-value pairs of the map to be garbage collected when the key is no longer referenced outside the WeakHashMap. It is beneficial to use WeakHashMap when you have to maintain registry-like data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

Sorting

To add support for Sorting, many changes have been done to the core Java libraries with the addition of Collection Framework in the Java 2 SDK version 1.2. Comparable interface can be implemented by classes like String and Integer so that a natural sorting order is provided. You can implement the Comparator interface to define your own order when you desire a different order than the natural order, or for the classes without a natural order.

To take advantage of the sorting capabilities, SortedSet and SortedMap are the two interfaces provided by Collections Framework.

- **Comparable Interface**

The Comparable interface, in the java.lang package, is for those classes, which have a natural ordering. The interface allows you to order the collection into that natural ordering when a collection of objects is of the same type.

- int compareTo(Object element)

compareTo(): Using this method, current instance can be compared with an element that is passed as an argument. If the current instance comes before the argument in the ordering, it returns a negative value, whereas if the current instance comes after, then a positive value is returned. Otherwise, it returns a zero. It is does not mean that a zero return value signifies equality of elements but it just signifies that two objects are ordered at the same position.

Several classes implement the Comparable interface. You will observe from the below table that some classes share the same natural ordering. Only mutually comparable classes can be sorted. This goes for the current release of the SDK. (that means the same class.)

The following table shows their natural ordering.

Class	**Ordering**
BigDecimal BigInteger	Numerical

Byte Double Float Integer Long Short	
Character	Numerical by Unicode value
CollationKey	Locale-sensitive string ordering
Date	Chronological
File	Numerical by Unicode value of characters in fully-qualified, system-specific pathname
ObjectStreamField	Numerical by Unicode value of characters in name
String	Numerical by Unicode value of characters in string

The documentation for the compareTo() method of String defines the ordering lexicographically. This implies the comparison is of the numerical values of the characters in the text, which is not necessarily alphabetically in all languages. For locale-specific ordering, use **Collator** with **CollationKey**.

The following demonstrates the use of Collator with CollationKey to do a locale-specific sorting:

```
import java.text.*;
import java.util.*;

public class CollatorTest {
  public static void main(String args[]) {
    Collator collator =
      Collator.getInstance();
    CollationKey key1 =
      collator.getCollationKey("Som");
    CollationKey key2 =
      collator.getCollationKey("som");
    CollationKey key3 =
      collator.getCollationKey("shon");
    CollationKey key4 =
```

```
      collator.getCollationKey("Shon");
    CollationKey key5 =
      collator.getCollationKey("Shonar");
    Set set = new TreeSet();
    set.add(key1);
    set.add(key2);
    set.add(key3);
    set.add(key4);
    set.add(key5);
    printCollection(set);
  }
  static private void printCollection(
      Collection collection) {
    boolean first = true;
    Iterator iterator = collection.iterator();
    System.out.print("[");
    while (iterator.hasNext()) {
      if (first) {
        first = false;
      } else {
        System.out.print(", ");
      }
      CollationKey key =
        (CollationKey)iterator.next();
      System.out.print(key.getSourceString());
    }
    System.out.println("]");
  }
}
```

Running the program produces the following output:

[shon, Shon, Shonar, som, Som]

Making your own class Comparable is just a matter of implementing the compareTo() method. It usually involves relying on the natural ordering of several data members. Your own classes should also override equals() and hashCode() to ensure two equal objects return the same hash code.

- **Comparator Interface**

When java.lang.Comparable cannot be implemented with a class or if you don't like the default Comparable behavior, you can provide your own java.util.Comparator.

- int compare(Object element1, Object element2)
- boolean equals(Object object)

The return values of both the methods, compare() method of Comparator and compareTo() method of Comparable are similar. In this case, if the first element comes before the second element in the ordering, it returns a negative value. If the first element comes after, then a positive value is returned. It returns a zero value. As we have seen in the case of Comparable, even in comparator interface, zero return value does not signify equality of elements. It just signifies that two objects are ordered at the same position.

It depends on the user of the Comparator to determine how to deal with it. If two unequal elements compare to zero, you should first be sure what you want. When used with a TreeSet or TreeMap it can be tedious to use a Comparator that is not compatible to equals(). With a Set, only the first will be added. With a map, the value for the second will replace the value for the second (keeping the key of the first).

To demonstrate, you may find it easier to write a new Comparator that ignores case, instead of using Collator to do a locale-specific, case-insensitive comparison. The following is one such implementation:

```
class CaseInsensitiveComparator implements
  Comparator {
    public int compare(Object element1,
      Object element2) {
        String lowerE1 = (
          (String)element1).toLowerCase();
        String lowerE2 = (
          (String)element2).toLowerCase();
        return lowerE1.compareTo(lowerE2);
  }
}
```

Since every class subclasses Object at some point, you need not implement the equals() method. In most cases you do not implement it. The equals() method compares only the comparator implementations and not the objects being compared.

With the Collections class, a predefined Comparator is available for reuse. Objects that implement the Comparable interface are sorted in reverse order by a comparator which is returned as a result of calling Collections.reverseOrder().

- **SortedSet Interface**

For maintaining elements in a sorted order, a special Set interface, SortedSet is provided by the Collections Framework.

- Comparator comparator()
- Object first()
- SortedSet headSet(Object toElement)
- Object last()
- SortedSet subSet(Object fromElement , Object toElement)
- SortedSet tailSet(Object fromElement)

Access methods are provided to the ends of the set as well as to subsets of the set by the interface. When you work with these subsets of the list, you can see that the changes to the subset are reflected in the source set and vice versa. This works because elements identify the subsets at the end points and not indices. Moreover, if the fromElement is part of the source set then it is also a part of the subset. However, if the toElement is part of the source set, it is not part of the subset. If you want a particular to-element to be in the subset, you have to find out the next element. In the case of a String, the next element is the same string with a null character appended (string+"\0").

The elements added to a SortedSet must either implement Comparable or you must provide a Comparator to the constructor of its implementation class: TreeSet. (You can implement the interface yourself. But the Collections Framework only provides one such concrete implementation class.)

To demonstrate, the following example uses the reverse order Comparator available from the Collections class:

```
import java.text.*;
import java.util.*;

public class Comp {
  public static void main(String args[]) {
    Comparator comparator = Collections.reverseOrder();
Set reverseSet = new TreeSet(comparator);
reverseSet.add("January");
reverseSet.add("February");
reverseSet.add("March");
reverseSet.add("February");
reverseSet.add("April");
System.out.println(reverseSet);
  }
}
```

Running the program produces the following output:

[March, January, February, April]

Because sets must hold unique items, if comparing two elements when adding an element results in a zero return value (from either the compareTo() method of Comparable or the compare() method of Comparator), then the new element is not added. If the elements are equal, then that is okay. However, if they are not, then you should modify the comparison method such that the comparison is compatible with equals().

Using the following creates a set with three elements: thom, Thomas, and Tom, not five elements as might be expected.

```
Comparator comparator =
  new CaseInsensitiveComparator();
Set set = new TreeSet(comparator);
set.add("Tom");
set.add("tom");
set.add("thom");
set.add("Thom");
set.add("Thomas");
```

- **SortedMap Interface**

SortedMap: It is a special Map interface provided by the Collections Framework for maintaining keys in a sorted order.

- Comparator comparator()
- Object firstKey()
- SortedMap headMap(Object toKey)
- Object lastKey()
- SortedMap subMap (Object fromKey, Object toKey)
- SortedMap tailMap (Object fromKey)

Access methods are provided by the interface to the ends of the map as well as to subsets of the map. A SortedMap workd just like a SortedSet, except that the sort is done on the map keys. TreeMap is the implementation class provided by the Collections Framework.

Since the key value pairs are unique, (i.e. one maps can only have one value for every key), if a zero value is returned while comparing two keys when adding a key-value pair (from either the compareTo() method of Comparable or the compare() method of Comparator), then new value replaces the value for the original key. If the result is zero then it is fine. But when they are not, then comparison method should be modified such that the comparison is compatible with equals().

Summary

In this chapter, you have learnt about:

- Collections provide a well-defined set of interfaces and classes to store and manipulate groups of data as a single unit. This unit is called a collection.

- The List and Set interfaces extend from the Collection interface.

- Features on which the Implementations of Collections Interfaces depend are :Performance, Ordered/Sorted, Uniqueness of items, Synchronized.

- The **iterator()** method of the **Collection** interface returns an **Iterator** class type.

- While creating the Collections Framework, to keep the design simple, the interfaces define all the methods an implementation class may provide.

- The **List** interface extends the **Collection** interface to define an ordered collection.

- The Map interface starts off its own interface hierarchy, for maintaining key-value associations.

- To implement sorting, SortedSet and SortedMap are the two interfaces provided by Collections Framework.

- Situations where some of the original collections capabilities rather than the new ones are to be used,