CHAPTER 19

COLLECTION INTERFACES

19.1 COLLECTION API INTERFACES

A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. It is a framework that provides architecture to store and manipulate the group of objects.

The Java Collections API's provide Java developers with a set of classes and interfaces that makes it easier to handle collections of objects. In a sense collection's works a bit like arrays, except their size can change dynamically, and they have more advanced behavior than arrays. All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

The Java collections framework standardizes the way in which groups of objects are handled by your programs. Prior to Java 2, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulates groups of objects. The collections framework was designed to meet several goals.

1. The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
2. The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
3. Extending and/or adapting a collection had to be easy.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the Collections class. Thus, they are available for all collections. The algorithms provide a standard means of manipulating collections. Another item created by the collections framework is the Iterator interface. An iterator gives you 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.

19.2 THE COLLECTION INTERFACES

The collections framework defines several interfaces.

Interface	Description
collection	Enables you to work with groups of objects; it is at the top of the Collections
hierarchy	List Extends Collection to handle sequences (lists of objects)
set	Extends Collection to handle sets, which must contain unique elements
SortedSet	Extends Set to handle sorted sets

19.2.1 Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

Vector is synchronized, and it contains many legacy methods that are not part of the collections framework. With the release of Java 2, Vector was reengineered to extend AbstractList and implement the List interface, so it now is fully compatible with collections.

Here are the **Vector** constructors:

Vector( ) - creates a default vector, which has an initial size of 10

Vector(int size) - creates a vector whose initial capacity is specified by size

Vector(int size, int incr) - creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.

Vector(Collection c) - creates a vector that contains the elements of collection c. This constructor was added by Java 2.

All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place. This reduction is important, because allocations are costly in terms of time.

The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don't specify an increment, the vector's size is doubled by each allocation cycle.

Vector defines these protected data members:

```
int capacityIncrement;
int elementCount;
Object elementData[ ];
```

The increment value is stored in capacityIncrement. The number of elements currently in the vector is stored in elementCount. The array that holds the vector is stored in elementData.

List of Vector methods are as below:

Method	Description
void addElement(Object element)	The object specified by element is added to the vector.
int capacity( )	Returns the capacity of the vector.
Object clone( )	Returns a duplicate of the invoking vector.
boolean contains(Object element)	Returns **true** if element is contained by the vector, and return false if it is not.
void copyInto(Object array[ ])	The elements contained in the invoking vector are copied into the array specified by array.
Object elementAt(int index)	Returns the element at the location specified by index.
Enumeration elements( )	Returns an enumeration of the elements in the vector.
void ensureCapacity(int size)	Sets the minimum capacity of the vector to size.
Object firstElement( )	Returns the first element in the vector.
int indexOf(Object element)	Returns the index of the first occurrence of element. If the object is not in the vector, -1 is returned.

int indexOf(Object element, int start)	Returns the index of the first occurrence of element at or after start. If the object is not in that portion of the vector, –1 is returned.
void insertElementAt (Object element, int index)	Adds element to the vector at the location specified by index.
boolean isEmpty( )	Returns true if the vector is empty and returns false if it contains one or more elements.
Object lastElement( )	Returns the last element in the vector.
int lastIndexOf(Object element)	Returns index of the last occurrence of element. If the object is not in the vector, -1 is returned.
int lastIndexOf(Object element, int start)	Returns the index of the last occurrence of element before start .If the object is not in that portion of the vector, -1 is returned.
void removeAllElements( )	Empties the vector. After this method executes, the size of the vector is zero.
boolean removeElement(Object element)	Removes element from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns true if successful and false if the object is not found.
void removeElementAt(int index)	Removes the element at the location specified by index.
void setElementAt(Object element,int index)	The location specified by index is assigned element.
void setSize(int size)	Sets the number of elements in the vector to size. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, null elements are added.
int size( ) String toString( ) void trimToSize( )	Returns the number of elements currently in the vector. Returns the string equivalent of the vector. Sets the vector's capacity equal to the number of elements that it currently holds.

Because **Vector** implements **List**, you can use a vector just like you use an **ArrayList** instance. You can also manipulate one using its legacy methods. For example, after you instantiate a **Vector**, you can add an element to it by calling **addElement( )**. To obtain the element at a specific location, call **elementAt( )**. To obtain the first element in the vector, call **firstElement( )**. To retrieve the last element, call **lastElement ( )**. You can obtain the index of an element by using **indexOf( )** and **lastIndexOf( )**. To remove an element, call **removeElement( )** or **removeElementAt( )**.

The following program uses a vector to store various types of numeric objects. It demonstrates several of the legacy methods defined by Vector. It also demonstrates the Enumeration interface.

```
// Demonstrate various Vector operations.
import java.util.*;
class VectorDemo
{
        public static void main(String args[]) {
                // initial size is 3, increment is 2
        Vector v = new Vector(3, 2);
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " + v.capacity());
```

```
    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));
    System.out.println("Capacity after four additions:
                        " + v.capacity());
    v.addElement(new Double(5.45));
    System.out.println("Current capacity: " + v.capacity());
    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));
    System.out.println("Current capacity: " + v.capacity());
    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));
    System.out.println("Current capacity: " + v.capacity());
    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
    System.out.println("First element: " +
                       (Integer)v.firstElement());
    System.out.println("Last element: " +
                       (Integer)v.lastElement());
    if(v.contains(new Integer(3)))
    System.out.println("Vector contains 3.");
                  // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();
    System.out.println("\nElements in vector:");
    while(vEnum.hasMoreElements())
    System.out.print(vEnum.nextElement() + " ");
    System.out.println();
    }
}
```

The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.
Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

19.2.2 Stack

LIFO

Stack is a subclass of Vector that implements a standard last-in, first-out stack. Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

The push() method pushes an object onto the top of the Stack.
The peek() method returns the object at the top of the Stack, but leaves the object on of the Stack.
The pop() method returns the object at the top of the stack, and removes the object from the Stack.
Apart from the methods inherited from its parent class Vector, Stack defines following methods:

boolean empty()

Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.

Object peek( )

Returns the element on the top of the stack, but does not remove it.

Object pop( )

Returns the element on the top of the stack, removing it in the process.

Object push(Object element)

Pushes element onto the stack. element is also returned.

int search(Object element)

Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

Example:

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

```
import java.util.*;

public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);
   }

   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

  public static void main(String args[]) {
    Stack st = new Stack();
```

```
        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);
        try {
        showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("empty stack");
            }
        }
    }
```

This would produce the following result:

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

Searching the Stack

You can search for an object on the stack to get it's index, using the `search()` method. The object's `equals()` method is called on every object on the `Stack` to determine if the searched-for object is present on the `Stack`. The index you get is the index from the top of the `Stack`, meaning the top element on the `Stack` has index 1.

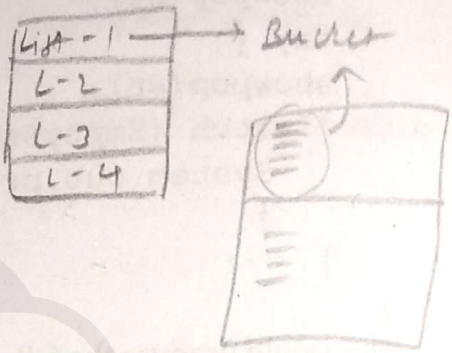
Here is how you search a `Stack` for an object:

```
Stack stack = new Stack();

stack.push("1");
stack.push("2");
stack.push("3");

int index = stack.search("3");      //index = 1
```

19.2.3 Hashtable

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. However, Java 2 re-engineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

A Hashtable is an array of list.Each list is known as a bucket.The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key. It implements the Map interface and extends Dictionary class.



- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

Example of Hashtable:

```
import java.util.*;
class TestCollection16{
   public static void main(String args[]){

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

   hm.put(100,"Amit");
   hm.put(102,"Ravi");
   hm.put(101,"Vijay");
   hm.put(103,"Rahul");

   for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
   }
  }
}
```

Output:

```
103 Rahul
      102 Ravi
      101 Vijay
      100 Amit
```

Construstors

- `Hashtable( )`

The Hashtable defines four constructors. The first version is the default constructor:

- `Hashtable(int size)`

The second version creates a hash table that has an initial size specified by size:

- `Hashtable(int size, float fillRatio)`

The third version creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.

This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.

- `Hashtable(Map m)`

The fourth version creates a hash table that is initialized with the elements in m. The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used.

Methods

Apart from the methods defined by Map interface, Hashtable defines the following methods:

void clear( )

Resets and empties the hash table.

Object clone( )

Returns a duplicate of the invoking object.

boolean contains(Object value)

Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.

boolean containsKey(Object key)

Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found.

boolean containsValue(Object value)

Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.

Enumeration elements( )

Returns an enumeration of the values contained in the hash table.

Object get(Object key)

Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.

boolean isEmpty( )

Returns true if the hash table is empty; returns false if it contains at least one key.

Enumeration keys( )

Returns an enumeration of the keys contained in the hash table.

Object put(Object key, Object value)

Inserts a key and a value into the hash table. Returns null if key isn't already in the hash table; returns the previous value associated with key if key is already in the hash table.

void rehash( )

Increases the size of the hash table and rehashes all of its keys.

Object remove(Object key)

Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned.

int size( )

Returns the number of entries in the hash table.

String toString( )

Returns the string equivalent of a hash table.

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;

public class HashTableDemo {

   public static void main(String args[]) {
      // Create a hash map
      Hashtable balance = new Hashtable();
      Enumeration names;
      String str;
      double bal;

    balance.put("Zara", new Double(3434.34));
    balance.put("Mahnaz", new Double(123.22));
    balance.put("Ayan", new Double(1378.00));
    balance.put("Daisy", new Double(99.22));
    balance.put("Qadir", new Double(-19.08));

    // Show all balances in hash table.
    names = balance.keys();
    while(names.hasMoreElements()) {
       str = (String) names.nextElement();
       System.out.println(str + ": " +
       balance.get(str));
    }
    System.out.println();
    // Deposit 1,000 into Zara's account
    bal = ((Double)balance.get("Zara")).doubleValue();
    balance.put("Zara", new Double(bal+1000));
    System.out.println("Zara's new balance: " +
    balance.get("Zara"));
    }
}
```

This would produce the following result:

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

Zara's new balance: 4434.34
```

19.2.4 Enumerations

An enum is a type that has a fixed list of possible values, which is specified when the enumis created. In some ways, an enum is similar to the boolean data type, which has true and false as its only possible values. However, boolean is a primitive type, while an enum is not.

The definition of an enum types has the (simplified) form:

```
enumhenum-type-namei{ hlist-of-enum-valuesi}
```

You can place it outside the main() routine of the program. The enum-type-name I can be any simple identifier. This identifier becomes the name of the enum type. Each value in the hlist-of-enum-values I must be a simple identifier, and the identifiers in the list are separated by commas. For example, here is the definition of an enum type named Season whose values are the names of the four seasons of the year:

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

Enum values are not variables. Each value is a constant that always has the same value. In fact, the possible values of an enum type are usually referred to as enum constants.

Note that the enum constants of type Season are considered to be "contained in" Season, which means- following the convention that compound identifiers are used for things that are contained in other things- the names that you actually use in your program to refer to the mare Season.SPRING, Season.SUMMER, Season.FALL, andSeason.WINTER.

Once an enum type has been created, it can be used to declare variables in exactly the same ways that other types are used. For example, you can declare a variable named vacation of type Season with the statement:

```
Season vacation;
```

After declaring the variable, you can assign a value to it using an assignment statement. The value on the right-hand side of the assignment can be one of the enum constants of type Season.Remember to use the full name of the constant, including "Season"!

For example:

```
vacation = Season.SUMMER;
```

You can print out an enum value with an output statement such as System.out.print(vacation).The output value will be the name of the enum constant (without the "Season."). In this case,the output would be "SUMMER".

Because an enum is technically a class, the enum values are technically objects. As objects, they can contain methods.

ordinal().

it returns the ordinal number of the value inthe list of values of the enum. The ordinal number simply tells the position of the value in the list.

```
Season.SPRING.ordinal() is the int value 0
Season.SUMMER.ordinal() is 1
Season.FALL.ordinal() is 2
Season.WINTER.ordinal() is 3.
```

Methods

boolean hasMoreElements( )

When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.

Object nextElement( )

This returns the next object in the enumeration as a generic Object reference.

Example:

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester {
   public static void main(String args[]) {
      Enumeration days;
      Vector dayNames = new Vector();
      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();
      while (days.hasMoreElements()){
      System.out.println(days.nextElement());
      }
   }
}
```

Output

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

19.2.5 The Set Interface

The Set interface defines a set. It extends Collection and declares the behavior of a collection that does not allow duplicate elements. Therefore, the add( ) method returns false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own.

The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. In addition to those methods defined by Set, the SortedSet interface declares some of the methods. Several

methods throw a **NoSuchElementException** when no items are contained in the invoking set. A **ClassCastException** is thrown when an object is incompatible with the elements in a set. A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the set. **SortedSet** defines several methods that make set processing more convenient. To obtain the first object in the set, call **first( )**. To get the last element, use **last( )**. You can obtain a subset of a sorted set by calling **subSet( )**, specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use **headSet( )**. If you want the subset that ends the set, use **tailSet( )**.

Methods

add( )

Adds an object to the collection

clear( )

Removes all objects from the collection

contains( )

Returns true if a specified object is an element within the collection

isEmpty( )

Returns true if the collection has no elements

iterator( )

Returns an Iterator object for the collection which may be used to retrieve an object

remove( )

Removes a specified object from the collection

size( )

Returns the number of elements in the collection

Example:

```
import java.util.*;

public class SetDemo {

 public static void main(String args[]) {
   int count[] = {34, 22,10,60,30,22};
   Set<Integer> set = new HashSet<Integer>();
   try{
     for(int i = 0; i<5; i++){
       set.add(count[i]);
     }
     System.out.println(set);

     TreeSet sortedSet = new TreeSet<Integer>(set);
     System.out.println("The sorted list is:");
     System.out.println(sortedSet);
```

```
            System.out.println("The First element of the set is: "+
                               (Integer)sortedSet.first());
            System.out.println("The last element of the set is: "+
                               (Integer)sortedSet.last());
        }
        catch(Exception e){}
    }
}
```

Output

```
java SetDemo
[34, 30, 60, 10, 22]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

19.2.6 The List Interface

The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. In addition to the methods defined by **Collection, List** defines some of its own methods. Several of these methods will throw an **UnsupportedOperationException** if the collection cannot be modified, and a **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. Methods defined in list interface are listed below :

Method	Description
void add(int index, Object obj)	Inserts obj into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int index, Collection c)	Inserts all elements of c into the invoking list at the index passed in index. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
Object get(int index)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object obj)	Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
int lastIndexOf(Object obj)	Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
ListIterator listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator listIterator(int index)	Returns an iterator to the invoking list that begins at the specified index.
Object remove(int index)	Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
Object set(int index, Object obj)	Assigns obj to the location specified by index within the invoking list.
List subList(int start, int end)	Returns a list that includes elements Elements in the returned list are also referenced by the invoking object.

Example:

```
import java.util.*;

public class CollectionsDemo {

  public static void main(String[] args) {
    List a1 = new ArrayList();
    a1.add("Zara");
    a1.add("Mahnaz");
    a1.add("Ayan");
    System.out.println(" ArrayList Elements");
    System.out.print("\t" + a1);

    List l1 = new LinkedList();
    l1.add("Zara");
    l1.add("Mahnaz");
    l1.add("Ayan");
    System.out.println();
    System.out.println(" LinkedList Elements");
    System.out.print("\t" + l1);
  }
}
```

Output

```
ArrayList Elements
        [Zara, Mahnaz, Ayan]
LinkedList Elements
        [Zara, Mahnaz, Ayan]
```

19.2.7 Map Interface

A map is an object that stores associations between keys and values, or key/value pairs. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.

The Map Interfaces

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map.
SortedMap	Extends Map so that the keys are maintained in ascending order.

The **Map** interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key. Several methods throw a **NoSuchElementException** when no items exist in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the map. An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object k)	Returns **true** if the invoking map contains k as a key. Otherwise, returns **false.**
boolean containsValue(Object v)	Returns **true** if the map contains v as a value. Otherwise, returns **false.**
Set entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry.** This method provides a set-view of the invoking map.
boolean equals(Object obj)	Returns **true** if obj is a **Map** and contains the same entries. Otherwise, returns **false.**
Object get(Object k)	Returns the value associated with the key k.
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty.Otherwise, returns **false.**
Set keySet( )	Returns a **Set** that contains the keys in theinvoking map. This method provides a set-view of the keys in the invoking map.
Object put(Object k, Object v)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map m)	Puts all the entries from m into this map.
Object remove(Object k)	Removes the entry whose key equals k.
int size( )	Returns the number of key/value pairs in the map.
Collection values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

There are two basic operations: **get( )** and **put( )**. To put a value into a map, use **put( )**, specifying the key and the value. To obtain a value, call get( ), passing the key as an argument. The value is returned. Maps are not collections because they do not implement the **Collection** interface, but you can obtain a collection-view of a map. To do this, you can use the entrySet( ) method. It returns a **Set** that contains the elements in the map. To obtain a collection-view of the keys, use **keySet( ).** To get a collection-view of the values, use values( ). Collection-views are the means by which maps are integrated into the collections framework.

The SortedMap Interface

The **SortedMap** interface extends **Map.** It ensures that the entries are maintained in ascending key order. Several methods throw a **NoSuchElementException** when no items are in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a null object when null is not allowed in the map.

Sorted maps allow very efficient manipulations of submaps (in other words, a subset of a map). To obtain a submap, use **headMap( ), tailMap( ), or subMap( ).** To get the first key in the set, call **firstKey( ).** To get the last key, use **lastKey( ).**

Method	Description
Comparator comparator( )	Returns the invoking sorted map's comparator. If the natural ordering is used for the invoking map, null is returned.
Object firstKey( )	Returns the first key in the invoking map.
SortedMap headMap(Object end)	Returns a sorted map for those map entries with keys that areless than end.
Object lastKey( )	Returns the last key in the invoking map.
SortedMap subMap(Object start, Object end)	Returns a map containing those entries with keys that are greater than or equal to start and less than end.
SortedMap tailMap(Object start)	Returns a map containing those entries with keys that are greater than or equal to start.

The Map.Entry Interface

The Map.Entry interface enables you to work with a map entry.

Method	Description
boolean equals(Object obj)	Returns **true** if obj is a **Map.Entry** whose key and value are equal to that of the invoking object.
Object getKey( )	Returns the key for this map entry.
Object getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
Object setValue(Object v)	Sets the value for this map entry to v. A **ClassCastException** is thrown if v is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with v. A NullPointerException is thrown if v is null and the map does not permit null keys. An **UnsupportedOperationException** is thrown if the map cannot be changed

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the Map interface.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

Notice that **AbstractMap** is a superclass for all concrete map implementations. **WeakHashMap** implements a map that uses "weak keys," which allows an element in a map to be garbage-collected when its key is unused.

Example:

```
import java.util.*;

public class CollectionsDemo {

   public static void main(String[] args) {
     Map m1 = new HashMap();
     m1.put("Zara", "8");
     m1.put("Mahnaz", "31");
     m1.put("Ayan", "12");
     m1.put("Daisy", "14");
     System.out.println();
     System.out.println(" Map Elements");
     System.out.print("\t" + m1);
   }
}
```

Output

```
Map Elements
     {Mahnaz=31, Ayan=12, Daisy=14, Zara=8}
```

19.2.8 Iterator class

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

Iterator class

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

ListIterator Class

Class	Description
void add(Object obj)	Inserts obj into the list in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
Object next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.

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

Using an Iterator

Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns **true.**
3. Within the loop, obtain each element by calling **next( ).**

For collections that implement **List,** you can also obtain an iterator by calling **ListIterator which** gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator.**

Here is an example that implements these steps, demonstrating both **Iterator** and **ListIterator.** It uses an **ArrayList** object, but the general principles apply to any type of collection. Of course, **ListIterator** is available only to those collections that implement the **List** interface.

```
// Demonstrate iterators.
import java.util.*;

class IteratorDemo
{
   public static void main(String args[])
   {
            // create an array list
      ArrayList al = new ArrayList();
            // add elements to the array list
      al.add("C");
      al.add("A");
      al.add("E");
      al.add("B");
      al.add("D");
      al.add("F");
            // use iterator to display contents of al
      System.out.print("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 iterated
        ListIterator litr = al.listIterator();
        while(litr.hasNext())
        {
            Object element = litr.next();
            litr.set(element + "+");
        }
        System.out.print("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.print("Modified list backwards: ");
        while(litr.hasPrevious())
        {
            Object element = litr.previous();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

The output is shown here:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```